Day 1

Assign	Laksh Pujary

I wilk skip the basic installation and hello world stuff.

Lets get to the extended Day 1:

✅ What We’re Building

1. ARP scan → find nearby devices
2. Port scan (1–1024) on each device
3. Ping each IP → Guess OS based on TTL

STEP 1: Setup ARP scan with `scapy`

```
from scapy.all import ARP, Ether, srp

def arp_scan(subnet="192.168.1.0/24"):
    print(f"[*] Scanning subnet: {subnet}")
    # ARP request packet to broadcast
```

```
    arp = ARP(pdst=subnet)
    ether = Ether(dst="ff:ff:ff:ff:ff:ff")
    packet = ether/arp

    result = srp(packet, timeout=2, verbose=0)[0]

    hosts = []
    for sent, received in result:
        hosts.append(received.psrc)

    print(f"[+] Found {len(hosts)} active hosts.")
    return hosts
```

📚 GLOSSARY OF TERMS

Term	Meaning
`Scapy`	A powerful Python library used to send, sniff, and manipulate network packets
`ARP`	Address Resolution Protocol — used to ask “Who has this IP address?” and get their MAC address
`Ether`	Ethernet layer — lets you set things like destination MAC address
`srp()`	Send and receive packets at Layer 2 (Ethernet) — returns sent/received pairs
`subnet`	A group of IPs. `192.168.1.0/24` means: “Scan all 256 IPs from .1 to .254”
`ff:ff:ff:ff:ff:ff`	A “broadcast” MAC address — it says: “Hey EVERYONE on this network, listen up”
`packet = ether/arp`	Combines the Ethernet and ARP layers into one packet
`timeout`	How long to wait for a reply before giving up
`verbose=0`	Don’t print logs while sending packets (quiet mode)
`psrc`	The IP address from the ARP reply (“who sent this response”)

Now let’s dive in deep:

`from scapy.all import ARP, Ether, srp`

We're telling Python:

> “Hey, I want to use Scapy — specifically the tools for making ARP packets (ARP), Ethernet frames (Ether), and a function to send/receive them (srp).”

- `ARP` = creates the "who has this IP?" question
- `Ether` = sets up the "send this question to everyone" part
- `srp()` = sends it out, waits for replies

`def arp_scan(subnet="192.168.1.0/24"):`

We're making a **function** called `arp_scan` that:

- Takes one input: a subnet (defaults to your home network range)
- Will return a list of IPs that replied

This is how you organize logic to reuse it.

`print(f"[*] Scanning subnet: {subnet}")`

We’re just giving feedback to the user:

> “Hey, I’m scanning this range of IPs”

The `f""` is an f-string, letting you insert variables inside the string.

`arp = ARP(pdst=subnet)`

We're creating the **ARP packet**:

- `pdst` = “protocol destination” = IP addresses to ask “Who are you?”

This says:

> “I want to know who owns every IP in this subnet.”

`ether = Ether(dst="ff:ff:ff:ff:ff:ff")`

We're making the **Ethernet frame** around the ARP packet.

- `ff:ff:ff:ff:ff:ff` is a broadcast address = shout to the whole network.

This line is like saying:

> “Wrap my ARP question in a loud envelope so everyone hears it.”

`packet = ether/arp`

We're stacking layers — this is how Scapy works.

We’re saying:

> “Send this Ethernet frame which contains an ARP request inside it.”

This is like nesting letters in an envelope.

`result = srp(packet, timeout=2, verbose=0)[0]`

This is where magic happens:

- `srp()` sends the packet
- Waits up to 2 seconds for replies
- Doesn’t show any logs (`verbose=0`)
- It returns a list of `(sent, received)` pairs

→ The `[0]` is just grabbing the first part of that tuple

This is like saying:

> “Send out my question. Tell me who responded, and forget about who didn’t.”

`hosts = [received.psrc for sent, received in result]`

This is a **list comprehension** — a compact way to build a list.

- `received.psrc` = the IP address of the responder

So this says:

> “For every reply, extract the IP address and save it in a list called hosts.”

`print(f"[+] Found {len(hosts)} active hosts.")`

Simple log:

> “How many devices replied?”

The `[+]` is a convention in CLI tools — like:

- `[+]` = positive result
- `[-]` = error
- `[*]` = info

`return hosts`

We're sending that list of IPs back to whoever called the `arp_scan()` function.

That means later you can do:

```
python
Copy code
devices = arp_scan()
```

And `devices` will be a list like:

```
python
Copy code
['192.168.1.1', '192.168.1.10', '192.168.1.30']
```

SUMMARY:

- **This function finds all devices** currently online in my network by shouting:

  “Who’s at 192.168.1.X?” and listening for replies
- It uses **ARP**, which works at the local network level (Layer 2)
- It returns a list of **IP addresses that replied**

```
import socket

def scan_ports(ip, ports=range(1, 1025)):
    print(f"\n[*] Scanning ports on {ip}...")
    open_ports = []
    for port in ports:
        try:
            sock = socket.socket()
            sock.settimeout(0.5)
            sock.connect((ip, port))
            open_ports.append(port)
            sock.close()
        except:
```

```
            pass
    return open_ports
```

📚GLOSSARY OF TERMS

Term	Meaning
`socket`	A Python module for network communication
`port`	A “door” into a computer (e.g., port 80 = HTTP)
`socket.socket()`	Opens a new network connection attempt
`settimeout(0.5)`	Don’t wait forever — give up after 0.5 seconds
`connect()`	Try to connect to that IP and port
`open_ports.append()`	If it responds, add it to our list
`sock.close()`	Always close your socket after you're done
`try...except`	A way to avoid crashing if something fails

Let’s explore line by line:

`import socket`

This pulls in Python’s built-in networking library — it lets you talk to ports.

> You’re basically saying:
>
> “Yo Python, I need to poke some ports. Give me low-level access to the internet stack.”

`def scan_ports(ip, ports=range(1, 1025)):`

You're defining a function named `scan_ports`.

- `ip` = the target computer (like `192.168.1.10`)
- `ports=range(1, 1025)` = you're saying:

> “By default, scan ports 1 through 1024.”

> In security, these are called well-known ports — used for common services like:
>
> - 22 → SSH
> - 80 → HTTP
> - 443 → HTTPS

---

print(f"\n[*] Scanning ports on {ip}...")

Just tells the user:

> “Hey, I’m checking this IP now.”

The `\n` puts a blank line before it for visual clarity.

---

open_ports = []

This creates an **empty list**.

We’ll add each open port we discover into this list.

---

for port in ports:

We’re going to loop over each number from 1 to 1024 (unless you specify something else).

It’s like saying:

> “Let’s try every possible door and see if it opens.”

---

try:

We’re about to run risky code.

If anything goes wrong (e.g., the port is closed, the host rejects us), this lets us **handle failure gracefully**.

`sock = socket.socket()`

This creates a new **socket object** — think of it as a "handshake attempt".

You're telling your OS:

> "Please open a connection channel for me."

`sock.settimeout(0.5)`

By default, sockets can hang forever.

This says:

> "Wait max 0.5 seconds before giving up."

It prevents your script from freezing.

`sock.connect((ip, port))`

This is where the actual connection is attempted.

If the port is **open**, it **won't error**.

If the port is **closed**, you'll get a `ConnectionRefusedError`.

So:

> "Try to talk to this door. If someone answers, it's open."

`open_ports.append(port)`

If we got here, the connection **succeeded**.

So we add that port to our list of wins.

---

sock.close()

Very important: You clean up the socket (free the resource).

This avoids leaking too many open sockets in your system.

---

except: pass

This says:

> “If any error happens (like port is closed), do nothing and keep going.”

We’re not logging errors here to keep the output clean — just hunting for wins.

---

return open_ports

After all ports are checked, we return a list of the ones that were open.

E.g.,

```
[22, 80, 443]
```

This tells us:

> “Hey, this machine is running SSH, HTTP, and HTTPS!”

SUMMARY

- We wrote a function to try knocking on every port on a device.
- If it gets an answer (i.e. no error), the port is open.
- It stores all successful ports in a list and gives it back.

This is basically the beginning of **every port scanner ever made.**

STEP 3: Ping IP and Guess OS from TTL

```
from scapy.all import IP, ICMP, sr1

def guess_os(ip):
    pkt = IP(dst=ip)/ICMP()
    reply = sr1(pkt, timeout=1, verbose=0)
    if reply:
        ttl = reply.ttl
        if ttl >= 128:
            return "Windows"
        elif ttl >= 64:
            return "Linux"
        else:
            return "Unknown"
    return "No response"
```

📚 Glossary Terms

Term	Meaning
`IP()`	Creates an IP packet. We use it to set the destination (`dst=...`)
`ICMP()`	Internet Control Message Protocol – the type of packet used by `ping`
`sr1()`	Send one packet and wait for **one** response (like a ping)
`TTL`	Time To Live – every OS sets a default TTL when it sends packets. Windows ≈ 128, Linux ≈ 64

What Are We Trying to Do?

We're going to:

1. **Ping** a device using an ICMP packet
2. **Look at the TTL** value in the reply
3. Use that TTL to **guess the OS**:
    - TTL ≈ 128 → Probably Windows
    - TTL ≈ 64 → Probably Linux
    - Anything else → Meh, we can't tell

Line-by-Line Deep Dive

`from scapy.all import IP, ICMP, sr1`

We're pulling in Scapy tools again:

- `IP()` → lets us make an IP packet
- `ICMP()` → gives us a ping request
- `sr1()` → sends that packet and waits for exactly 1 reply

`def guess_os(ip):`

We're defining a function:

> "Give us an IP address, and we'll try to guess the OS."

`pkt = IP(dst=ip)/ICMP()`

We're building the packet:

- `IP(dst=ip)` → destination is the IP we want to ping
- `/ICMP()` → we're attaching an ICMP Echo Request to it

In Scapy, using `/` stacks layers — this is a complete ping packet now.

Think of it like:

> Envelope: IP
>
> Letter inside: ICMP
>
> Sent to: the target IP

`reply = sr1(pkt, timeout=1, verbose=0)`

We send the packet and wait for a reply.

- `timeout=1` → wait only 1 second
- `verbose=0` → don't show logs

If there's no response, `reply` will be `None`.

`if reply:`

If we *did* get a response, we keep going.

If not, we return "No response" (we'll get to that at the end).

`ttl = reply.ttl`

We pull the **TTL** value out of the response.

🧠 TTL = "How many hops this packet can survive before dying"

But **each OS sets a default TTL** when it sends a packet:

- Windows → usually starts at 128
- Linux → usually starts at 64

So this value **clues us in** to what OS the responder is using.

`if ttl >= 128: return "Windows"`

If TTL is 128 or more, we guess:

> "This is probably Windows."

`elif ttl >= 64: return "Linux"`

If it’s between 64 and 127, we guess:

> “Looks like Linux.”

`else: return "Unknown"`

If it’s anything weird or low — like 30, 55, 12, etc. — we can’t confidently say what OS it is.

`return "No response"`

This is the fallback.

If the host **didn’t respond at all**, we return:

> “No response” — might be offline, firewalled, or ignoring pings

---

🧠 Summary in Plain Words

- We build a ping packet
- We send it to the target
- If it responds, we check the TTL
- Based on that, we **infer the operating system**

It’s not 100% accurate, but it's a **lightweight way to fingerprint devices**.

STEP 4: Combine It All

```
def main():
    subnet = "192.168.1.0/24"
    hosts = arp_scan(subnet)

    for ip in hosts:
```

```
        ports = scan_ports(ip)
        os_guess = guess_os(ip)
        print(f"\n[+] {ip} - OS: {os_guess}")
        print(f"    Open Ports: {ports if ports else 'None'}")

if __name__ == "__main__":
    main()
```

Explanation – Line by Line

def main():

We're defining a function called `main()`.

This is our tool's **central brain**.

Just like in every team, there's a person who coordinates everything — `main()` is that person.

It will:

- Run the ARP scan
- Loop through each IP
- Run the port scan
- Guess the OS
- Print results

---

subnet = "192.168.1.0/24"

We're setting the network to scan.

- `192.168.1.0/24` is a common home Wi-Fi subnet
- `/24` means:

> "Scan all addresses from .1 to .254"

This line defines the **scope** of the scan.

`hosts = arp_scan(subnet)`

We call the ARP scanner from earlier and pass in our subnet.

- It returns a list of IPs that replied
- We store that list in a variable called `hosts`

At this point, we have **a list of live devices** on the network.

`for ip in hosts:`

We now **loop** through each IP address in that list.

> For example:
>
> - IP 1: `192.168.1.1`
> - IP 2: `192.168.1.10`
> - IP 3: `192.168.1.42`

We're going to scan each of them individually.

`ports = scan_ports(ip)`

We call our **port scanner**, sending it the IP address.

It returns a list of open ports (or an empty list).

Example:

```
python
Copy code
[22, 80, 443]
```

`os_guess = guess_os(ip)`

We call our **OS guessing function**.

- If we get a reply, it returns `"Windows"`, `"Linux"`, or `"Unknown"`

- If nothing replies, we get `"No response"`

We now know:

- Who is online
- Which ports are open
- What OS they might be running

`print(f"\n[+] {ip} - OS: {os_guess}")`

We show the results:

```
css
Copy code
[+] 192.168.1.10 - OS: Windows
```

The `\n` gives a blank line for better spacing between devices.

`print(f" Open Ports: {ports if ports else 'None'}")`

We print open ports.

If the list `ports` has data, we show it.

If it's empty, we just say `"None"`.

This keeps our output clean, readable, and compact.

`if __name__ == "__main__":`

This is Python's way of saying:

> “Only run main() if this file is being run directly.”

If someone imports our scanner into another script later, `main()` won’t run — which is great behavior for reusable tools.

`main()`

We finally run the main function — kicking off the entire flow.

🧠 Final Mental Picture

We built a tool that:

1. Looks around the local network
2. Says “Who’s here?”
3. Tries every port on each device
4. Pings them and reads the TTL
5. Outputs:
    - IP
    - OS
    - Open ports

That’s a full-blown, real-world **recon script** — the kind used by both security engineers and attackers.

FULL SCRIPT:

```
from scapy.all import ARP, Ether, srp, IP, ICMP, sr1
import socket

def arp_scan(subnet="192.168.1.0/24"):
    print(f"[*] Scanning subnet: {subnet}")
    arp = ARP(pdst=subnet)
    ether = Ether(dst="ff:ff:ff:ff:ff:ff")
    packet = ether/arp

    result = srp(packet, timeout=2, verbose=0)[0]
    hosts = [received.psrc for sent, received in result]
```

```
    print(f"[+] Found {len(hosts)} active hosts.")
    return hosts

def scan_ports(ip, ports=range(1, 1025)):
    print(f"\n[*] Scanning ports on {ip}...")
    open_ports = []
    for port in ports:
        try:
            sock = socket.socket()
            sock.settimeout(0.5)
            sock.connect((ip, port))
            open_ports.append(port)
            sock.close()
        except:
            pass
    return open_ports

def guess_os(ip):
    pkt = IP(dst=ip)/ICMP()
    reply = sr1(pkt, timeout=1, verbose=0)
    if reply:
        ttl = reply.ttl
        if ttl >= 128:
            return "Windows"
        elif ttl >= 64:
            return "Linux"
    return "Unknown"

def main():
    subnet = "192.168.1.0/24"
    hosts = arp_scan(subnet)

    for ip in hosts:
        ports = scan_ports(ip)
        os_guess = guess_os(ip)
        print(f"\n[+] {ip} - OS: {os_guess}")
```

```
        print(f"    Open Ports: {ports if ports else 'None'}")

if __name__ == "__main__":
    main()
```