# How to Whitelist IP Addresses to Access Desired Docker Containers?

Divya Mamgai Follow
Oct 28, 2020 · 7 min read

## Motivation

Recently I had to secure one of my docker setups running in a virtual machine so that only specific ports (or docker containers) are accessible via a specific set of IP addresses on the network. Now, this seems to be a simple task, and mind you it is, but not if you aren't well versed with iptables and how docker configures it for its network. Since I was unable to find any tutorial for this particular requirement I decided to write one.

***Disclaimer:*** *I'm quite new to the networking world of Linux and iptables in general, please let me know in the comments if there is an even easier way to achieve this and or any enclosed information is incorrect.*

## Exploration

Docker does have a documentation page on how to configure iptables with docker here. And it does outline a way to whitelist a single IP. So yay?… job done? Not quite. It only allows single IP to access **ALL** ports in the Docker network. Not quite what we need. If that is your use case I suggest you stop reading this any further and go through the documentation, finish your task at hand, and come back if you would like to know how to add restrictions on port level.

### Uncomplicated Firewall made complicated by docker?

UFW is an easier and faster way of managing firewalls rules in Linux. Well for one thing it's in the name—"uncomplicated". But as I found out with docker UFW is more like CFW or more colloquially known as TFW. I won't be getting into the details here but because of the way docker manipulates the iptables normal UFW flow won't work, in fact, it will cause more problems than it solves—learned that the hard way. There are lots of tutorials out there on how to get this to work and even a github project dedicated to extending UFW to support docker. If most of your setup already relies on UFW I suggest you check those out, since for my use-case it seems quite the overkill and I didn't bother exploring it.

## But what the heck are iptables?

Instead of me explaining what they are, which will be a bad job, you can look at the wikipedia page for iptables and very detailed information in the official documentation. The only particulars needed for this post are the following. Iptables mainly have four types of table — raw, mangle, nat, and filter. And they are processed in that order. For more granularity go through the diagram below.

## Understanding how docker manipulates iptables

Docker creates two custom iptables chains in the **filter** table named **DOCKER** and **DOCKER-USER**. And all of the communication to docker is validated by these rules. They recommend not changing the **DOCKER** chain since it is populated by docker networking modules. And prepend (not append) all of the custom validation you might require to the **DOCKER-USER** chain. The important piece of information which the docker's documentation page leaves out is that all of the port forwarding rules for the containers are added in the **nat** table of iptables, and this will be a big gotcha later. To see how the filter table is configured let's execute the following command.

```
sudo iptables -L -v
```

All of the incoming connections related to docker are routed to docker network interfaces, hence any filtering of these connections (or packets) will take place in the

**FORWARD** filter chain. Hence check the **FORWARD** chain, you will see that the first entry is **DOCKER-USER**. This means all (because there are no constraints added to the rule) of the packets received to this chain will be passed to the **DOCKER-USER** chain. So let's jump to the **DOCKER-USER** chain, there (by default) you should see just a single **RETURN** rule with no constraints, which means that filtering will start again from the next rule in the parent chain. If we don't want to process any packet we will have to filter it before the **RETURN** rule.

## Getting to the point

Let's recap our objective — only allow a specific ip to communicate with a specific service running on a specific port. You might need this if you have developed a black box that just takes an input and produces output without exposing any implementation details. You would probably want to block all access to the box apart from input and output streams. So let's say we have two docker containers running — `django` (at `8080`) and `postgres` (at `5432`). And we only want to allow incoming connections to the `django` server at `8080` and not allow any outside connections to the database.

## Don't allow any outside connections by default

To drop all incoming and forwarded connections by default but allow established connections and all ssh connections execute the following commands. We are assuming that there is only one external network interface ( `eth0` ) in the system.

```
# Allow loopback connections.
iptables -A INPUT -i lo -j ACCEPT
iptables -A OUTPUT -o lo -j ACCEPT

# Allow established and related incoming connections.
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j
ACCEPT

# Allow established outgoing connections.
iptables -A OUTPUT -m conntrack --ctstate ESTABLISHED -j ACCEPT

# Allow all incoming and outgoing SSH connections.
iptables -A INPUT -p tcp --dport 22 -m conntrack --ctstate
NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -p tcp --sport 22 -m conntrack --ctstate
ESTABLISHED -j ACCEPT

# Drop all incoming and forwarding connections by default.
iptables -P INPUT DROP
iptables -P FORWARD DROP

# Drop all forwarded connections from the external interface in the
DOCKER-USER chain.
iptables -I DOCKER-USER -i eth0 -j DROP
```

*For more information and reference for these commands visit this handy blog post over at DigitalOcean.*

So now if you again check the output of `sudo iptables -L -v` you will see that for **INPUT** and **FORWARD** chain default policy (shown in brackets after the name of the chain) is now set to **DROP**.

**INPUT** chain should look something like this.

```
Chain INPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source
destination
  555  416K ACCEPT     all  --  lo     any     anywhere
anywhere
  147 11595 ACCEPT     all  --  any    any     anywhere
```

```
anywhere             ctstate RELATED,ESTABLISHED
    1    44 ACCEPT     tcp  --  any    any     anywhere
anywhere             tcp dpt:ssh ctstate NEW,ESTABLISHED
```

**DOCKER-USER** chain should look something like this.

```
Chain DOCKER-USER (1 references)
 pkts bytes target     prot opt in     out     source
destination
    0     0 DROP       all  --  eth0   any     anywhere
anywhere
   16   864 RETURN     all  --  any    any     anywhere
anywhere
```

Now **all of the packets coming to this chain from the** `eth0` **interface will be dropped**.

## It's quite lonely here, let someone talk to me

Assuming that we have a static IP ( `192.168.0.69` ) from where we would like to send requests to our `django` server running at port `8080` . To allow these requests to be forwarded to the `django` container execute the following commands.

```
# Allow inbound and outbound traffic for 192.168.0.69 IP on 8080
port.
iptables -I DOCKER-USER -i eth0 -s 192.168.0.69 -p tcp --dport 8080 -
j RETURN
iptables -I DOCKER-USER -o eth0 -d 192.168.0.69 -p tcp --sport 8080 -
j RETURN
```

The above commands add two rules stating that any forwarded `tcp` packets either incoming or outgoing from `192.168.0.69` on port `8080` should **RETURN** from the **DOCKER-USER** chain to its parent chain (**FORWARD**) for further processing and skip any subsequent rules in this chain. **Hence those packets are not dropped since the dropping rule comes after them**. Now `django` server should be able to receive requests from that IP address. You can prepend as many rules as you want to allow different IP addresses or complete subnets. The job is done.

## But wait there's a gotcha?

Remember how I told you docker modifies the **nat** table of iptables as well? Docker handles all of the exposed ports from your containers using it. Let's see what exactly it does by running the following command.

```
sudo iptables -t nat -L DOCKER
```

You should see an output similar to this.

```
Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
DNAT       tcp  --  anywhere             anywhere             tcp
dpt:5432 to:172.18.0.12:5432
DNAT       tcp  --  anywhere             anywhere             tcp
dpt:8080 to:172.18.0.7:8080
DNAT       tcp  --  anywhere             anywhere             tcp
```

This allows all of the connections made on the exposed ports on the host network to get forwarded to the docker network and its corresponding container.

Do you see the problem here? Let me explain. Since the **nat** table is processed before the **filter** table in iptables, **if we have specified a different exposed port than the port of the process running inside the docker container** we will run into issues while writing our filtering rules.

Let's say in our example above if the `django` server was running on port `8080` but we exposed (and or mapped) it as `8081` , to not get any interference during development with other `django` based servers which might be running as well. In this case, if you were to run `sudo iptables -t nat -L DOCKER` you should see output similar to this.

```
Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere
DNAT       tcp  --  anywhere             anywhere             tcp
dpt:5432 to:172.18.0.12:5432
DNAT       tcp  --  anywhere             anywhere             tcp
dpt:8081 to:172.18.0.7:8080
DNAT       tcp  --  anywhere             anywhere             tcp
```

We would expect to use port `8081` while writing our IP whitelisting rule as described in the previous section. But that won't work since those packets would have already been forwarded to port `8080` and will violate the constraints of the whitelisting rule. Figuring this out made me scratch my head for a long while. Hence while writing IP whitelisting rules as mentioned above you should use the port number of the underlying docker container and not the host mapped port as you would expect. Even though you would access the `django` server using port `8081` on the whitelisted IP.

## And why should I care about this?

Cause it's secure.

## Update

If you want to allow outgoing connections from your docker services, like connecting to DataDog, AWS, GCP, and so on, you will have to allow established connections to pass through. This can be done by adding the following rule to the **DOCKER-USER** chain.

```
# Allow established and related incoming connections.
iptables -I DOCKER-USER -m conntrack --ctstate ESTABLISHED,RELATED -j
RETURN
```

Docker Networking Iptables Security Linux