Securing Linux - Advanced IPTables

29 Minutes Remaining

Instructions Resources Help  100%

Advanced IPTables

The first topic that we will cover is the structure of **iptables**. There are three tables: **filter**, **nat**, and **mangle**. The default table is **filter** and it is used for all packet filtering. The **nat** table is used for Network Address Translation, including IP and port forwarding. The **mangle** table is used for changing values in the packet headers, for example the **Type of Service** field.

There are also 5 builtin chains. These are **PREROUTING**, **INPUT**, **FORWARD**, **OUTPUT**, and **POSTROUTING**. Not all tables have all of these chains.

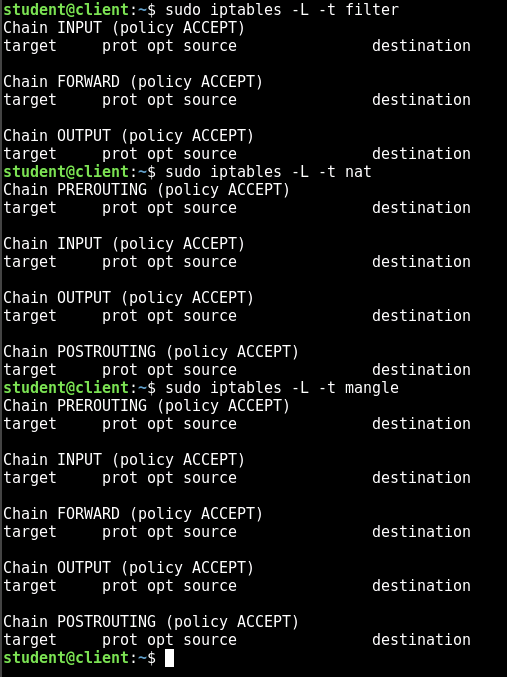
1. **View Default Chains**

To see which chains are present on each table, let's examine it.

sudo iptables -L -t filter

sudo iptables -L -t nat

sudo iptables -L -t mangle



The **-L** option lists the rules, and the **-t** option allows you to specify a table. The default table is **filter**, so the **-t filter** options is technically redundant, but included for consistency.

You can see from this output that the **filter** table has **INPUT**, **FORWARD**, and **OUTPUT**. The **nat** table has all of the chains except **FORWARD**, and **mangle** has all of them. These are processed in a specific order, which we will demonstrate in the next step. Generally speaking, the **PREROUTING** chain is for a packet that has just come in on an interface and the **POSTROUTING** chain is for packets just about to leave an interface. The **INPUT** chain is for packets going to the local machine, **OUTPUT** is for packets sent from the local machine, and **FORWARD** is for packets that are for another machine and passing from one interface to another. This last one is the case when the machine is acting as a router.

1. **See the Table and Chain Order**

To look at the order in which the chains are processed, we are going to add rules to each chain in each table that will log the packets. The log entry will be prefixed with the table and chain name, and then we will send packets to the machine, from the machine, and forwarded through the machine.

sudo iptables -t filter -A INPUT -j LOG --log-prefix filter-input

sudo iptables -t filter -A FORWARD -j LOG --log-prefix filter-forward

sudo iptables -t filter -A OUTPUT -j LOG --log-prefix filter-output

sudo iptables -t nat -A PREROUTING -j LOG --log-prefix nat-prerouting

sudo iptables -t nat -A INPUT -j LOG --log-prefix nat-input

sudo iptables -t nat -A OUTPUT -j LOG --log-prefix nat-output

sudo iptables -t nat -A POSTROUTING -j LOG --log-prefix nat-postrouting

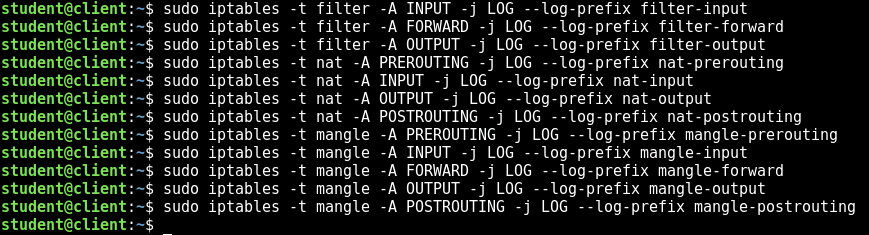
sudo iptables -t mangle -A PREROUTING -j LOG --log-prefix mangle-prerouting

sudo iptables -t mangle -A INPUT -j LOG --log-prefix mangle-input

sudo iptables -t mangle -A FORWARD -j LOG --log-prefix mangle-forward

sudo iptables -t mangle -A OUTPUT -j LOG --log-prefix mangle-output

sudo iptables -t mangle -A POSTROUTING -j LOG --log-prefix mangle-postrouting



The **-A** option appends a rule to the end of the specified chain. There is also a **-I** option that will insert a rule at a given position in the chain. The **-j** option allows you to specify a target. Common targets are **ACCEPT**, **DROP**, and **REJECT**.

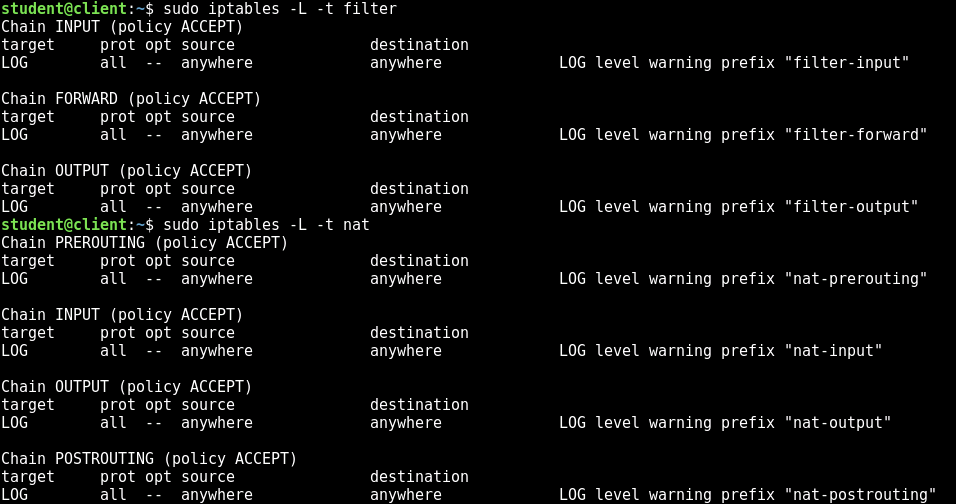
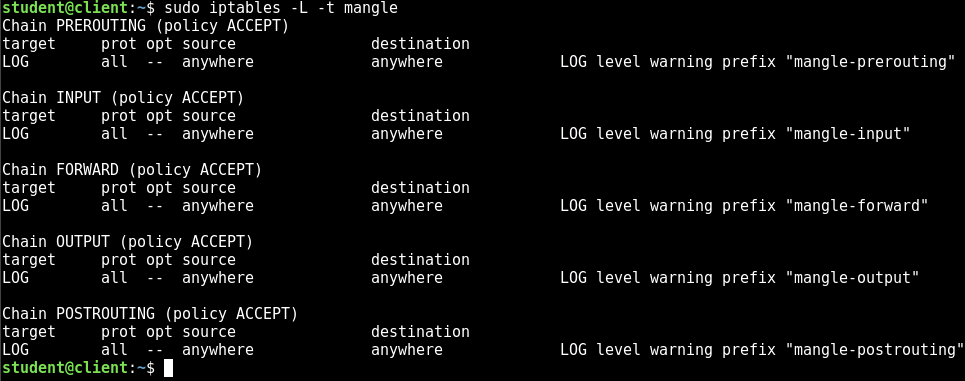
The **LOG** target is a special target. Most targets, when triggered, stop the processing of the packet. **LOG** targets don't stop the processing. They perform their action, and then **iptables** continues processing the rules in the chain. The **LOG** target also can take the additional options **--log-prefix** which let's you specify a string to be prepended to the log entry.

Now, let's inspect the rules.

sudo iptables -L -t filter

sudo iptables -L -t nat

sudo iptables -L -t mangle

Now let's test it out. Switch to the Fedora machine.

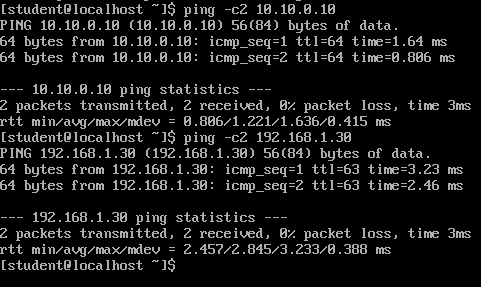
[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Login with the username **student** and the password **student**. Then, you will **ping** the Debian machine (10.10.0.10) and the Ubuntu machine (192.168.1.30)

The networking in this lab is configured so that the Debian machine is on two networks. The Fedora machine is on one of them (10.10.0.0/24) and the Ubuntu machine is on the other (192.168.1.0/24). The Debian machine is configured to route traffic between the two networks. This way, if we send a packet from the Fedora machine to the Ubuntu machine, it will get forwarded through Debian, and thus trigger the **FORWARD** chains.

ping -c2 10.10.0.10

ping -c2 192.168.1.30

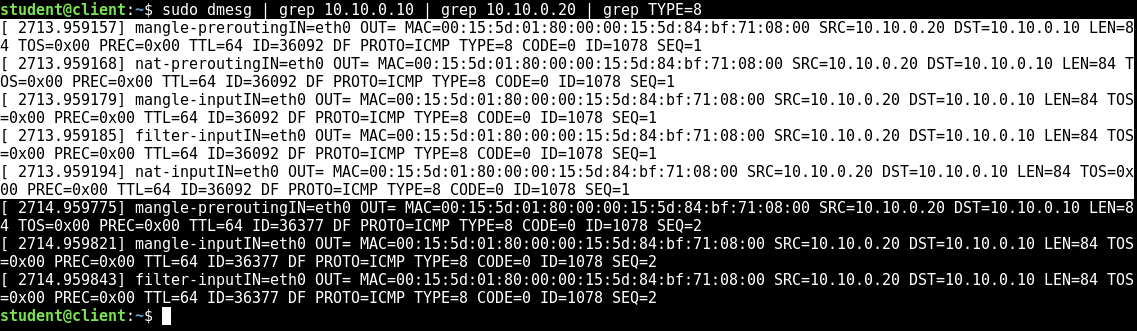


Now, switch back to the Debian machine.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

The logs are placed in the **dmesg** log file, which can be viewed with the command of the same name. We are going to filter the output so that we see just what we want.

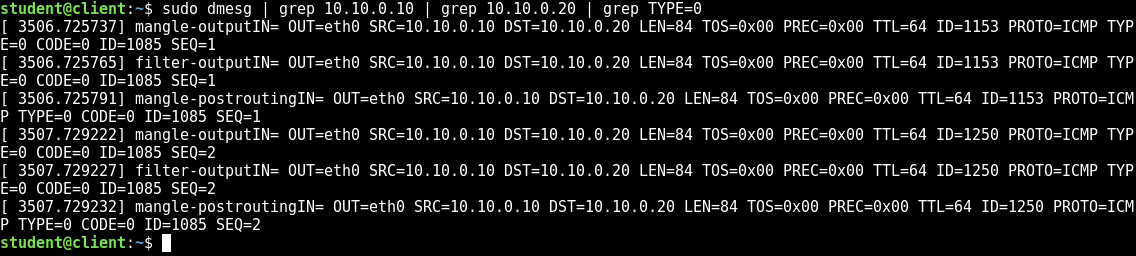
sudo dmesg | grep 10.10.0.10 | grep 10.10.0.20 | grep TYPE=8



What is shown here are the two incoming ping packets, with the first one highlighted. They both follow the same pattern, except that the second packet does not seem to go through the **nat** table. The **nat** table keeps track of state, and so it knows that these packets don't need to be processed. However, when taken all together, we can see that they are processed in the following order: Mangle Prerouting -> Nat Prerouting -> Mangle Input -> Filter Input -> Nat Input

From this, we can see that **mangle** comes first, **filter** next, and **nat** last. For the chains, **PREROUTING** comes first and then **INPUT**. Now, let's look at the responses.

sudo dmesg | grep 10.10.0.10 | grep 10.10.0.20 | grep TYPE=0

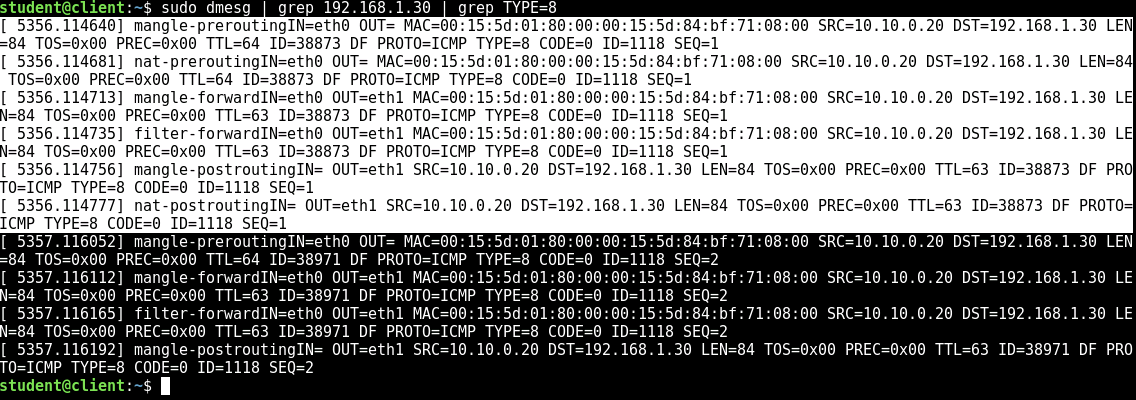


If you wait too long, the log messages may disappear from the **dmesg** log. If that happens, you can just switch back to Fedora and resend the pings.

For the outgoing packet, we see the same pattern, with **mangle** coming first, and then **filter**, and **OUTPUT** being processed before **POSTROUTING**.

Now, let's look at what happens for routed, or forwarded, packets. Again, resend the ping requests if needed.

sudo dmesg | grep 192.168.1.30 | grep TYPE=8



Again, we see the two packets, this time hitting **PREROUTING** first, then **FILTER**, and finally **POSTROUTING**. Also, each time, we see **mangle** first, then the other applicable chain, either **nat** or **filter**.

For more information on how the packets flow through the different tables and chains, see <https://danielmiessler.com/study/iptables/>.

The chains we discussed are the default, built in, chains. **iptables** also supports user-defined chains, which can aid in compartmentalizing and organizing the firewall rules.

PreviousNext: Setting up NAT

Live Chat

Securing Linux - Advanced IPTables

29 Minutes Remaining

Instructions Resources Help  100%

Setting up NAT

As the name implies, the **nat** table allows you to configure Linux to implement Network Address Translation (NAT). We will look at two different forms of this that can be performed by the **iptables** **nat** table, standard NAT and port forwarding.

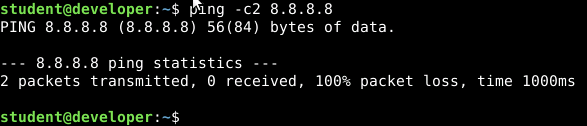
1. **Set Up NAT**

Right now, if we tried to ping from the Ubuntu machine to the IP address 8.8.8.8, it would not work. Switch to Ubuntu and try this out.

[Ubuntu](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Login with the username **student** and the password **student**. Launch a terminal and execute the following command.

ping -c2 8.8.8.8



The reason is that, even though IP forwarding is enabled on the Debian machine, when Ubuntu's packets get forwarded by Debian to the gateway on 10.10.0.1, it sees a packet from a private IP address, but one that is not on its subnet, and so it gets dropped. In order to get packets out to the internet from Ubuntu and get the responses back, we need to enable NAT on Debian.

The way NAT works is that it takes the incoming packet, and replaces the real source IP address (in this case 192.168.1.30) with its own IP address (10.10.0.10). This is what is sent out to the internet. In addition to performing this translation, it stores an entry in its internal NAT table, so that when the response comes back, it knows which machine was the original source, and can rewrite the packet again, this time replacing the destination IP address with 192.168.1.30.

Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Much of this is already built into **iptables**. The only thing we need to do is add the following single rule to the **nat** table's **POSTROUTING** chain.

sudo iptables -t nat -A POSTROUTING ! -d 10.10.0.0/24 -o eth0 -j SNAT --to-source 10.10.0.10

sudo iptables -t nat -L

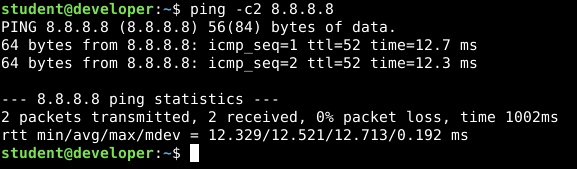


We are using a few extra options here. We have a few options to limit the application of the NAT only to packets going to the internet. First, we are checking that the destination is not on the subnet 10.10.0.0/24. This is with the negation **!** and the **-d** option. We also use the **-o** option to apply this rule only to packets that are going out on eth0, which is the interface connected to the internet. We then use the **SNAT** target, which causes most of the NAT logic to apply. This automatically applies to the responses coming back, even though we have not added a rule that includes that. Finally, the **SNAT** target takes an argument of **--to-source**, in which we list the address that we want overwritten in the place of the original source IP address. In this case, we use the IP address on the eth0 interface.

Now let's test it. Switch to Ubuntu.

[Ubuntu](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

ping -c2 8.8.8.8



This time, it works, and we are able to reach out to the internet.

1. **Port Forwarding**

One of the side effects of NAT is that it acts something like a firewall for the internal machine. In this case, that is the Ubuntu machine. Other computers that could access the Debian machine (for example, if Debian had a public IP address) would not be able to access the Ubuntu machine, even though the Ubuntu machine would, in fact, be able to access any external machine. However, if we wanted to run a server on Ubuntu and have it publicly accessible, you can achieve this with Port Forwarding.

If we enable port forwarding on Debian, the port that we wish to forward will be open on Debian. All packets sent to Debian's IP address on that port will be rewritten and forwarded to the Ubuntu machine on the port that we desire. This allows us to selectively enable access on certain ports. For this example, we are going to forward port 2222 on Debian to port 22 on Ubuntu. We will then be able to ssh from Fedora to port 2222 on Debian, which will actually connect us to the ssh server on Ubuntu.

Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Let's add the following rule to enable the port forward.

sudo iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 2222 -j DNAT --to 192.168.1.30:22

sudo iptables -t nat -L

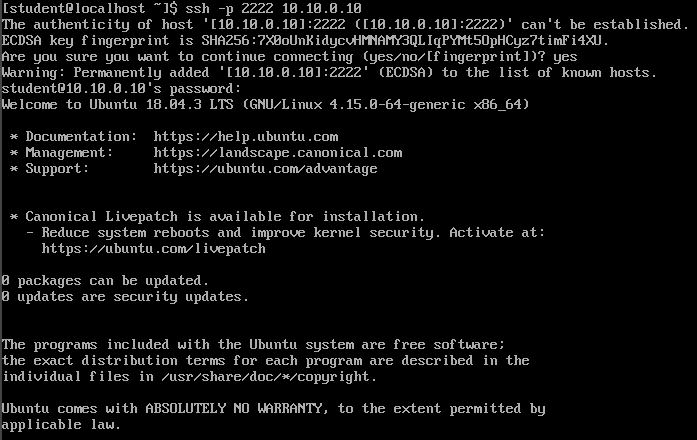


Now, let's test it. Switch to the Fedora machine.

[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Connect via ssh to port 2222 on 10.10.0.10.

ssh -p 2222 10.10.0.10



Enter **yes** to trust the key. Once connected, execute the command:

ip a



Notice that the IP address does not say 10.10.0.10, but rather says 192.168.1.30. This means that even though we connected to port 2222 on 10.10.0.10, we were forwarded on to the regular ssh port on Ubuntu.

Now exit the ssh session.

exit

redo-1

PreviousNext: Limits

Live Chat

Securing Linux - Advanced IPTables

28 Minutes Remaining

Instructions Resources Help  100%

Limits

So we have seen how **iptables** can be used to enable connectivity, now let's see how it can be used to limit connectivity. Simple packet filtering was already covered in a previous lab, and so in this section, we will look at how **iptables** can be used to impose limits on legitimate and allowed connections. For example, we want to allow external hosts to connect to Debian over ssh, but we want to limit the number of connection attempts that can be made in a given amount of time. This has the effect of being undetectable to normal users, who would connect once and have a long lived session, but would also have the effect of greatly slowing down an attacker who is trying to brute force the ssh login password.

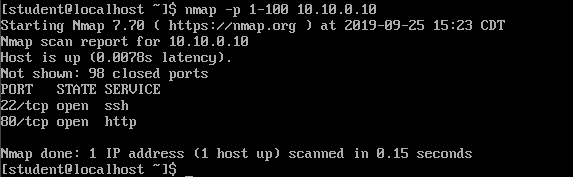
There are a number of different ways that rate limiting can be achieved. There is a **limit** module and a **hashlimit** module which work in similar ways but with a few differences. There is also protection built into the **conntrack** module. We will investigate these methods.

1. **Basic Rate Limiting**

First, let's look at basic rate limiting. The approach we will take in this step will simply limit the amount of packets that match the criteria. It uses the **limit** module which has basic limiting capabilities. We will also make use of a new chain to better organize the rules.

Before we add the rules, let's get a baseline for scanning speed. Still on Fedora, scan the Debian machine on ports 1 - 100.

nmap -p 1-100 10.10.0.10



At the bottom, it tells you how long it took to perform the scan. In the screenshot above, it took 0.15 seconds. Now, let's add the rate limiting rules to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

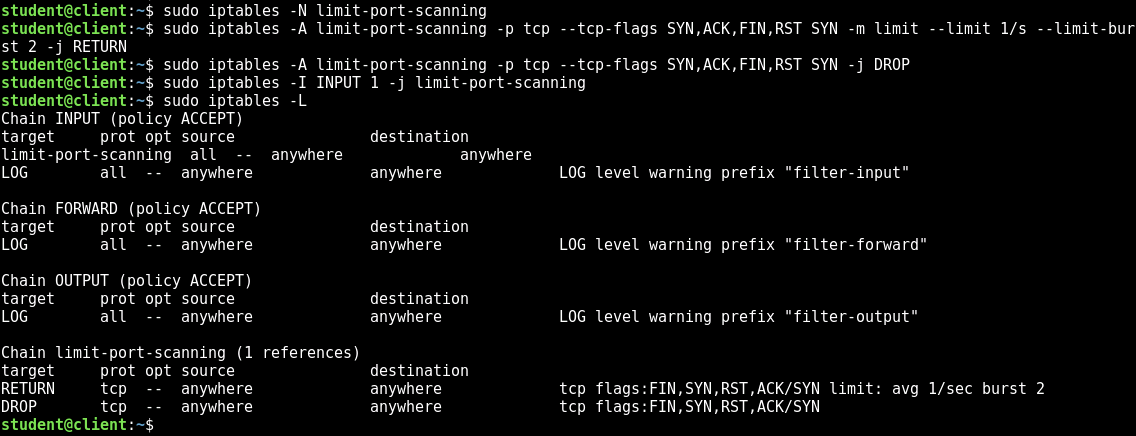
sudo iptables -N limit-port-scanning

sudo iptables -A limit-port-scanning -p tcp --tcp-flags SYN,ACK,FIN,RST SYN -m limit --limit 1/s --limit-burst 2 -j RETURN

sudo iptables -A limit-port-scanning -p tcp --tcp-flags SYN,ACK,FIN,RST SYN -j DROP

sudo iptables -I INPUT 1 -j limit-port-scanning

sudo iptables -L



What do these do? The rule in **INPUT** simply jumps over to the **limit-port-scanning** chain that we added. This chain has two rules. The first sets up the criteria for the rate limiting. If the packet is within the rate restrictions, it will match and the target **RETURN** is triggered. This stops processing on the **limit-port-scanning** chain and goes back to the **INPUT** chain. If, however, the packet is above the limits, then the rule does not match, and the following rule is processed, which simply drops all packets.

For the rate limiting rule, we used several options. First, we used the **--tcp-flags** option to indicate which TCP flags we wanted to match. There are two arguments here - both of which can be comma separated lists. The first is the set of TCP flags that we want to compare. The second is the list of TCP flags that should be set. The ones that are absent in the second list must be unset. So in this case, for the rule to match, the SYN flag must be set and the ACK, FIN, and RST flags must not be set. This will narrow the packets down to those trying to create new connections. Then we establish the limits. In this case, we are saying that there is a limit of 1 packet per second with a burst of 2 packets. This has the effect of limiting the flow to two packets per second.

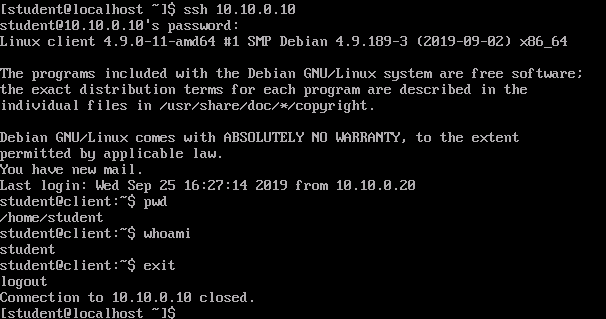
Let's test the effect now. Switch back to Fedora.

[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Scan the Debian machine again.

nmap -p 1-100 10.10.0.10

This time, it will take several minutes to complete. Feel free to terminate the scan early once you are convinced that the rate limiting is having the desired effect. However, normal connections will still work. Feel free to connect to 10.10.0.10 over ssh with the command **ssh 10.10.0.10**, logging in with the username **student** and the password **student**. Test out a few commands are see the responsiveness. Make sure to **exit** at the end of your testing.

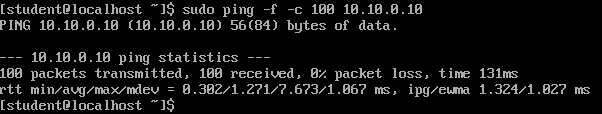


You could also combine the above rule with **conntrack** to apply it to **NEW** connections, instead of relying on TCP flags.

1. **Hashlimit**

Next, let's test out the **hashlimit** module. This has a lot of the same basic functionality of **limit**, but has more features and more expressiveness. We will demonstrate a few of the features, but you should read the manual to see the full extent of its capabilities. We will be limiting **ICMP** messages, so let's test the performance of the network before we implement the rules. Still in Fedora, run the following command.

sudo ping -f -c 100 10.10.0.10



This mode of **ping** is flood mode, in which it sends packets as fast as it can. You can see that it took a total time of 131 ms to send all 100 packets and none of the packets were lost. Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Add the following rule.

sudo iptables -A INPUT -p icmp -m hashlimit --hashlimit-name ICMPTEST --hashlimit-mode srcip --hashlimit-srcmask 32 --hashlimit-above 5/second --hashlimit-burst 2 --hashlimit-htable-expire 3000 -j DROP

sudo iptables -L



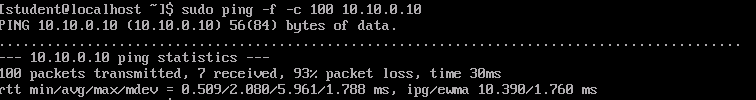
Let's unpack this rule. We are applying this rule only to ICMP packets. We are required to give the **hashlimit** table a name, which we call ICMPTEST in this case. We then specify that we want the rule to apply on a per source IP basis. This is expressed with the mode being set to **srcip** and the source mask being set to **32**. We then have to specify a limit. In this case, we are saying that the rule will match if the rate is more than 5 packets per second. We are also allowing a burst of 2. The last option, **--hashlimit-htable-expire** is how often the records for each IP are purged. This is expressed in milliseconds, so in this case, that is 3 seconds.

Let's test this new rule. Switch back to Fedora.

[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

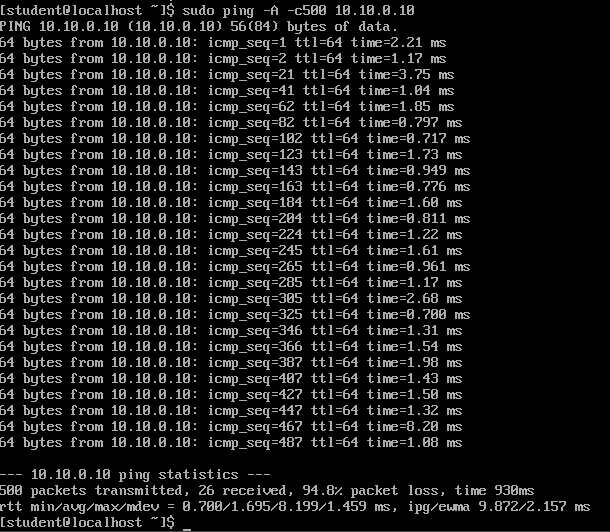
Rerun the ping flood and compare with the previous results.

sudo ping -f -c 100 10.10.0.10



We can see from these results that only 7 packets of the 100 got through, even though it took less time. We can get a better intuitive feel for how great the slowdown is if we try the following command. This will send 500 ICMP packets as fast as it can.

sudo ping -A -c500 10.10.0.10



We see that the first two come through immediately. This is the burst value. Then it is roughly every 20 packets that make it though.

1. **Recent**

In this step, we will use the **recent** module to limit ssh brute forcing speed. Again, let's see the performance we can get before we implement the rules. We still have the port scanning limit in place, so let's remove that first.

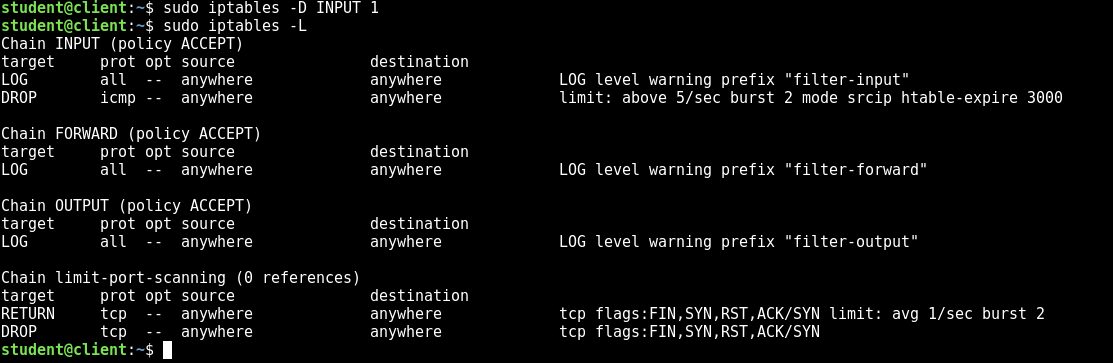
Switch to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Remove the rule to jump to the **limit-port-scanning** chain.

sudo iptables -D INPUT 1

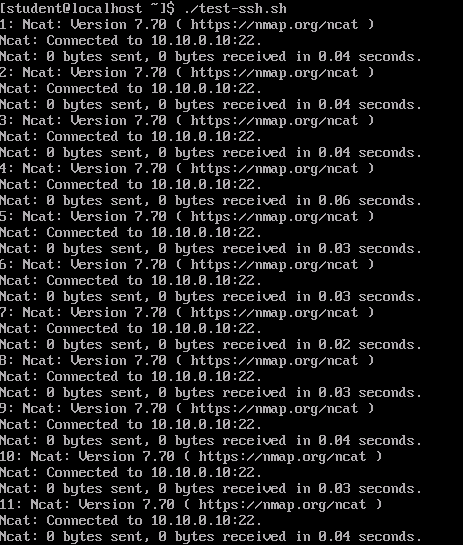
sudo iptables -L



Now switch to Fedora, and let's simulate an ssh brute force attack with a local script called **test-ssh.sh**. This custom script simply calls **nc -vz 10.10.0.10 22** 20 times.

[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

./test-ssh.sh



As we can see, all 20 connections completed immediately. Now switch back to Debian.

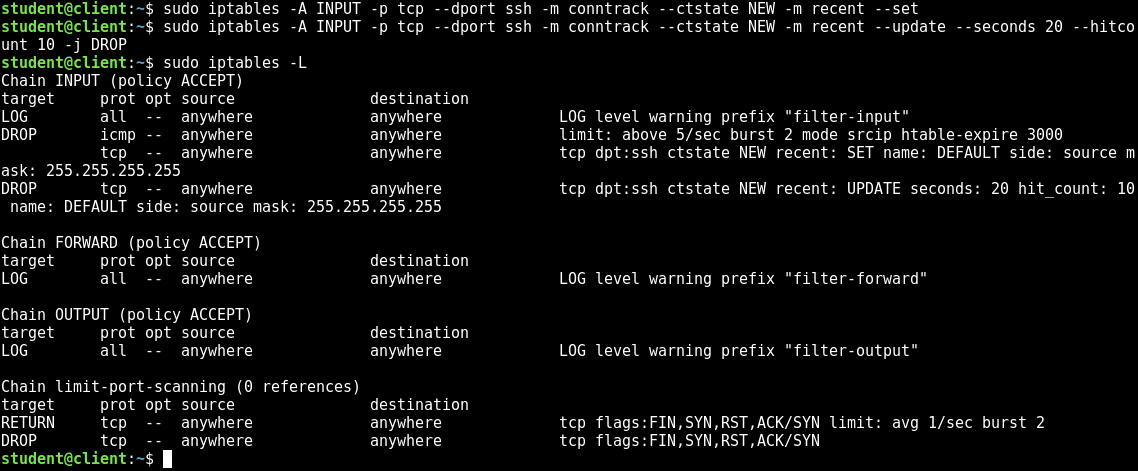
[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Add the following two rules.

sudo iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -m recent --set

sudo iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -m recent --update --seconds 20 --hitcount 10 -j DROP

sudo iptables -L

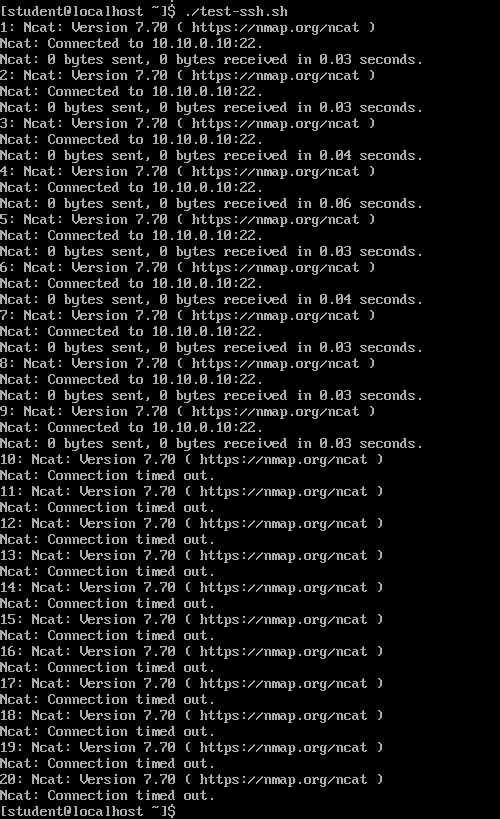


These two rules first make sure that the recent module is being used, and then, if the hit count is more than 10 new ssh connections in 20 seconds, it will drop the packet.

Now switch to Fedora so we can test it.

[Fedora Server](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

./test-ssh.sh



The first 9 connections succeed immediately, but once it hits the limit, it gets hung up. These connection attempts will start timing out and no further connections will succeed. This is effectively reducing the speed at which you can make new connections to the ssh server.

1. **Connlimit**

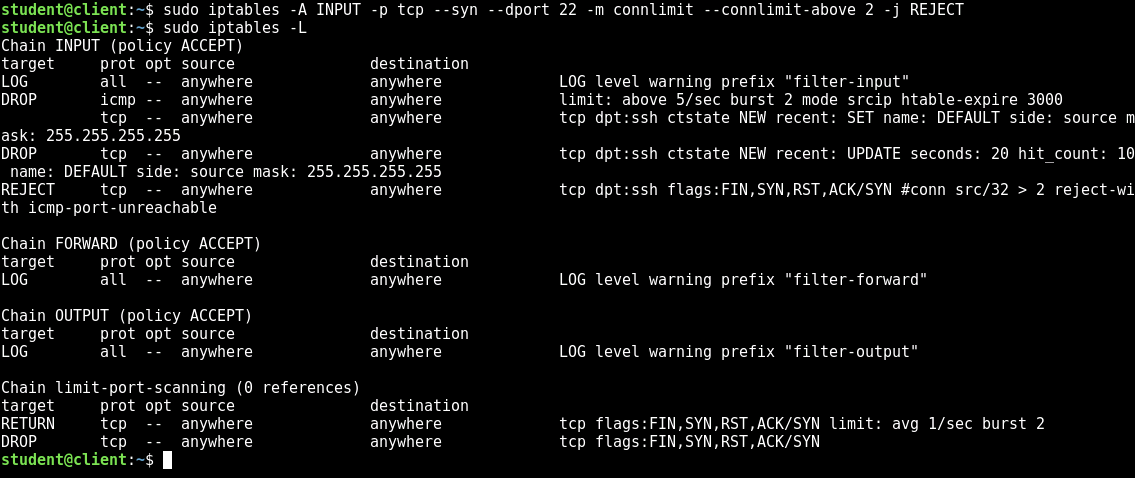
The last rate limiting module we will discuss is **connlimit**. This module is similar to the previous, but the difference is that its focus is on limiting parallel connections. That is, preventing hosts from having too many active connections at a given time. Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Add the following **iptables** rule.

sudo iptables -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 2 -j REJECT

sudo iptables -L

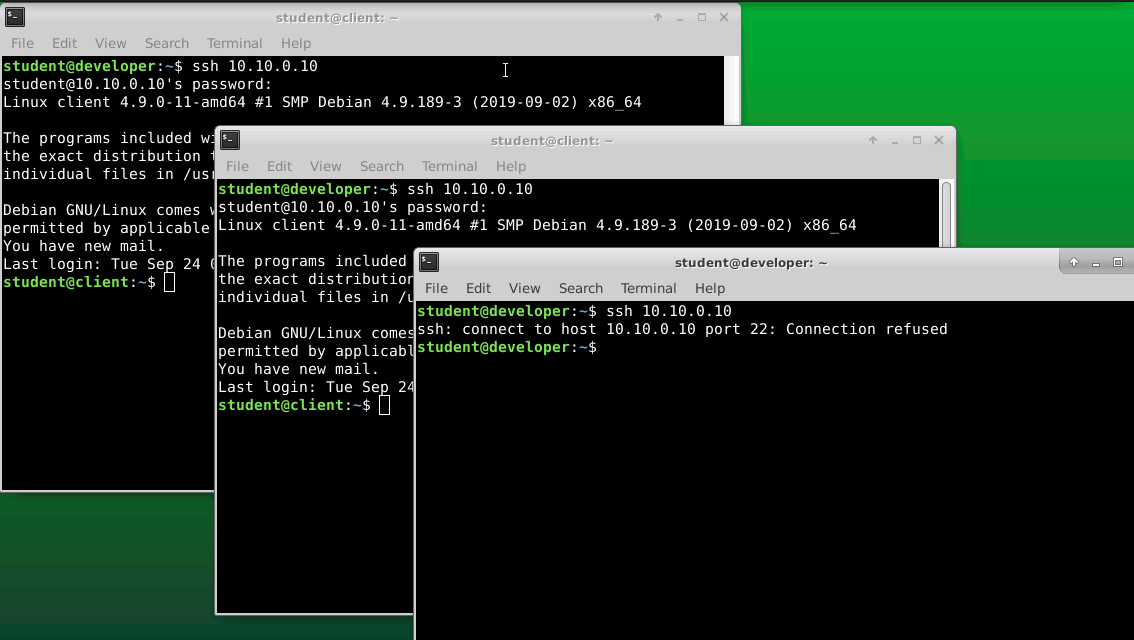


Now let's test it. Switch to Ubuntu.

[Ubuntu](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Open three terminal windows an in each one log in to 10.10.0.10 with ssh.

ssh 10.10.0.10



You can see that the first two connections succeeded, but the third one was rejected.

**Exit** out of the ssh sessions that connected.

PreviousNext: Blocking Malicious or Ill-Formed...

Live Chat

Securing Linux - Advanced IPTables

28 Minutes Remaining

Instructions Resources Help  100%

Blocking Malicious or Ill-Formed Packets

In addition to rate limiting you can also use **iptables** to block packets that you suspect are suspicious. This is particularly effective for blocking malformed packets, but can also be used for some simple content based filtering. Of course, for more complicated and robust content blocking, you would likely want to make use of an Intrusion Detection System. However, **iptables** can be a low overhead way of implementing some simple blocking.

1. **Block Packets With Bogus TCP Flags**

We have already seen that **iptables** rules can incorporate TCP flags. With this, we can list out rules that would block many of the invalid combinations. Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Add the following rules.

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG NONE -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,SYN FIN,SYN -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags SYN,RST SYN,RST -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,RST FIN,RST -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,ACK FIN -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,URG URG -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,FIN FIN -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,PSH PSH -j DROP

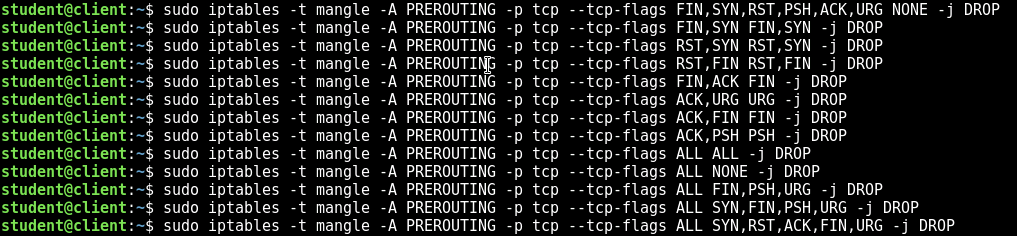
sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL ALL -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL NONE -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL FIN,PSH,URG -j DROP

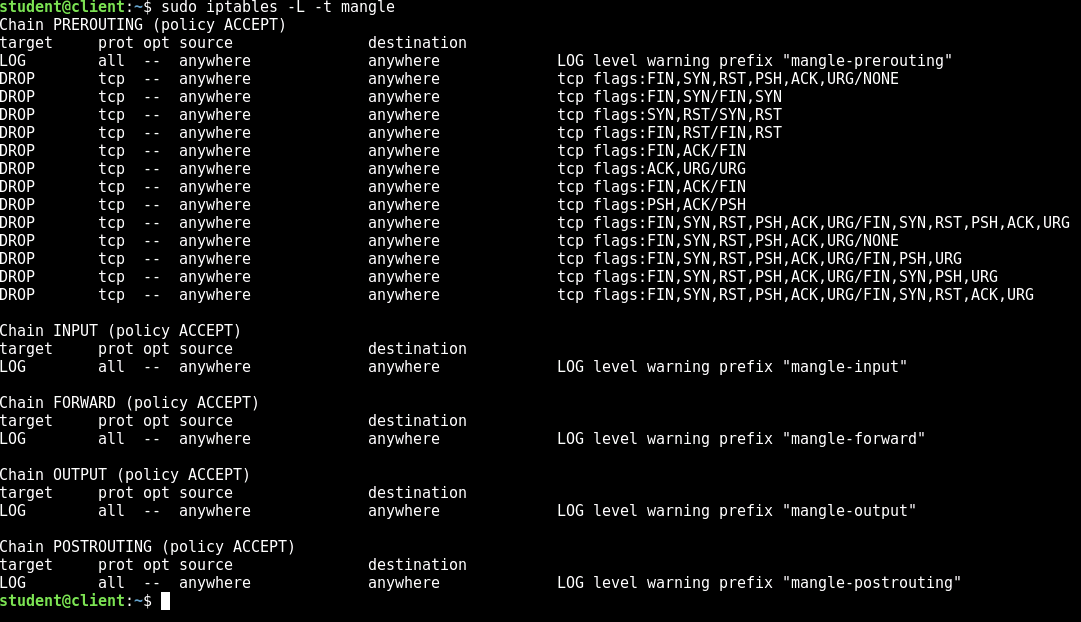
sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL SYN,FIN,PSH,URG -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL SYN,RST,ACK,FIN,URG -j DROP



Let's view them, before we move on.

sudo iptables -L -t mangle

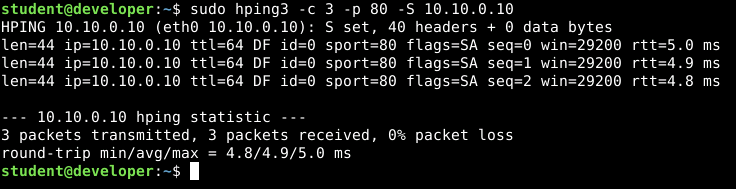


Now, let's test it out, using the tool **hping3**. **hping3** is a tool that allows you to create arbitrary packets. That means you can set custom flags that would otherwise never be seen on a network. Switch to Ubuntu.

[Ubuntu](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

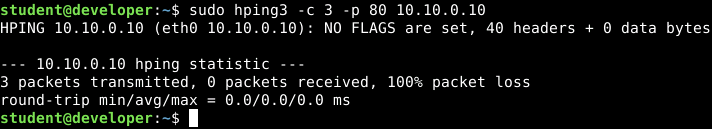
Try sending a few packets and see what happens. First, we will send 3 TCP packets to port 80 with just the SYN flag set.

sudo hping3 -c 3 -p 80 -S 10.10.0.10



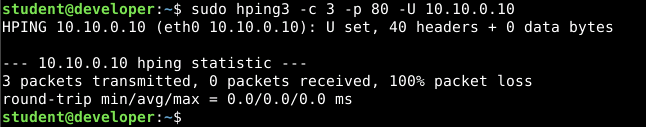
We can see that we get a response for each packet, each one having the SYN and ACK flags set, as indicated by the **flags=SA** in the response. Next, let's send packets with no TCP flags set.

sudo hping3 -c 3 -p 80 10.10.0.10



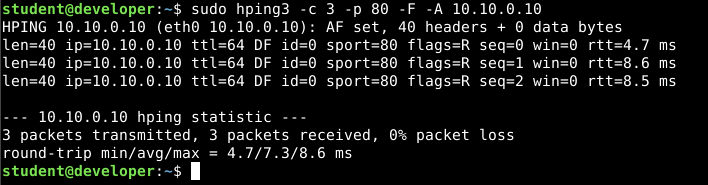
This time we can see that it says no TCP flags are set. We can also see that we got no responses. Let's try another combination. We will send packets with the URG flag set.

sudo hping3 -c 3 -p 80 -U 10.10.0.10



Finally, lets try a combination that should pass the firewall. We will send a packet with the FIN and ACK flags set.

sudo hping3 -c 3 -p 80 -F -A 10.10.0.10



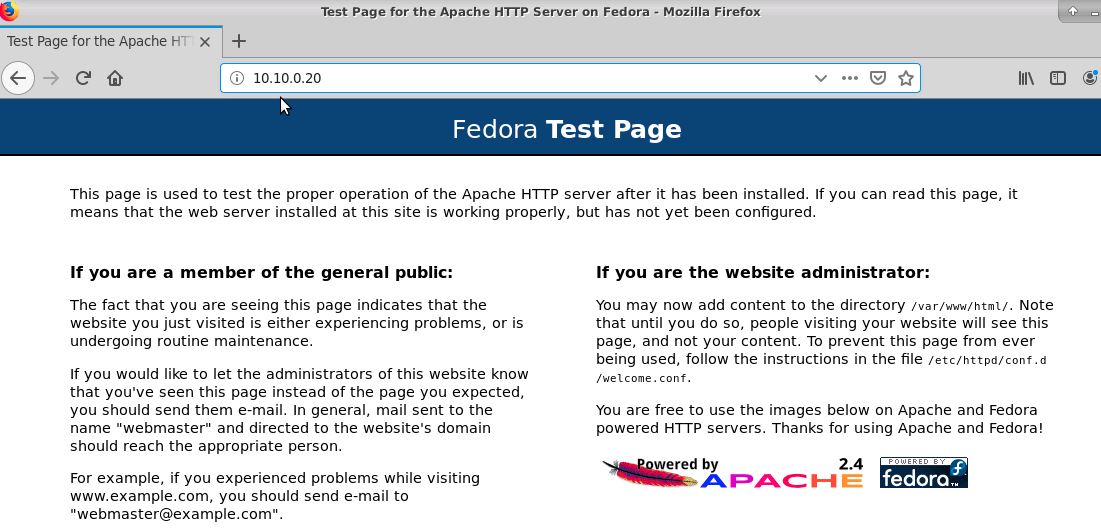
As you can see, we get responses for these packets. So the TCP flag filter is working.

We added these rules to the **mangle** table, in the **PREROUTING** chain. This means that when a packet comes in, the very first thing that happens is that it is checked for valid TCP flags and is dropped if they are not valid. This means that they don't even get considered by the later rules.

1. **Content Filtering**

In addition to all the other filtering we have done so far, **iptables** can also filter based on content. This can be done with the **string** and the **hex-string** options. Let's test this out.

Still in Ubuntu, open Firefox and navigate to **http://10.10.0.20**.

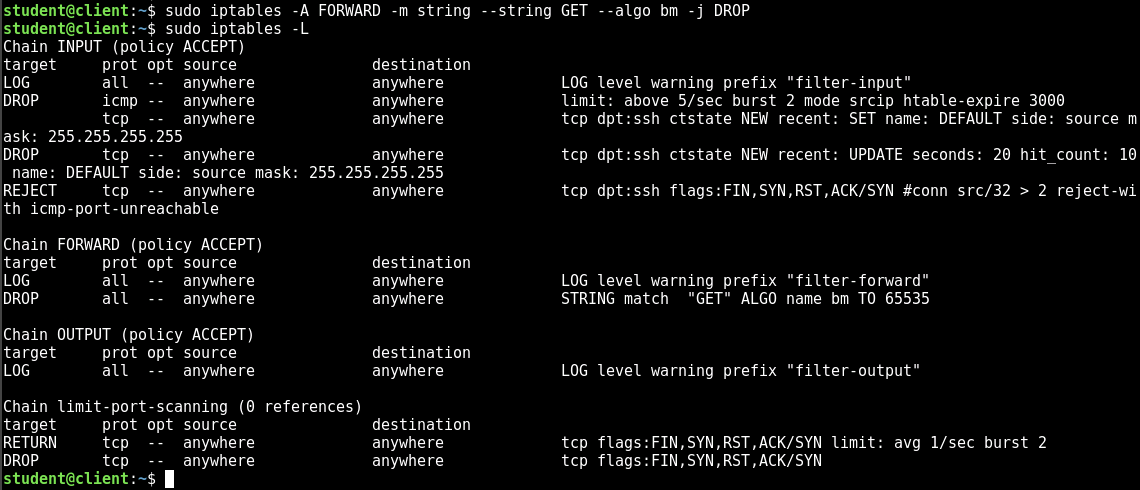


The page will load normally. Now switch to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

sudo iptables -A FORWARD -m string --string 'GET' --algo bm -j DROP

sudo iptables -L

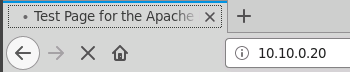


The only option here that is probably not obvious is the **--algo** option. This designates the pattern matching algorithm that we will use to find the string. There are two options here: Boyer-Moore (**bm**) and Knuth-Morris-Pratt (**kmp**). These algorithms have different pros and cons based on the scenario in which they will be used. For our purposes, it will not make a difference. For a production environment, it may be worth it to investigate the tradeoffs and potentially do some benchmark testing to see which would perform better.

Now, switch back to Ubuntu.

[Ubuntu](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

Reload the webpage in Firefox. The page will hang while trying to load. Eventually, it will timeout.



This is because even though the connection succeeds, the packet containing the **GET** request is dropped by Debian and never makes it to Fedora.

You can also use the **--hex-string** option to block based on raw binary content.

PreviousNext: Limiting By User and Group

Live Chat

Securing Linux - Advanced IPTables

28 Minutes Remaining

Instructions Resources Help  100%

Limiting By User and Group

The final type of filtering that we will do is by user or group. More fully featured firewalls can block network traffic based on which application is trying to make the connection. **iptables** cannot quite do that, but it can block based on the user or group that is running the application. Thus, you can lock down applications by creating special users or groups, and you can use **iptables** to create rules that apply specifically to them.

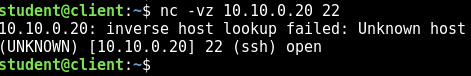
1. **Block by User or Group**

Switch back to Debian.

[Debian](https://labclient.labondemand.com/Instructions/cf709220-7824-49b5-b145-33584ec8b00c)

For comparison, let's first show that we can connect to 10.10.0.20 on port 22.

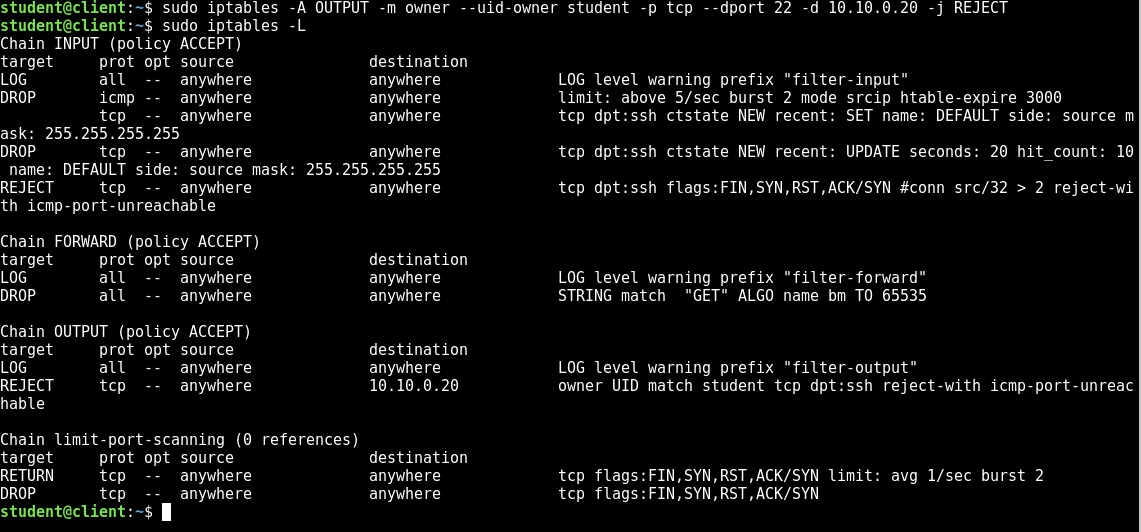
nc -vz 10.10.0.20 22



We can see that the connection succeeds. Now, let's limit the ability of the user **student** to connect to 10.10.0.20 on port 22.

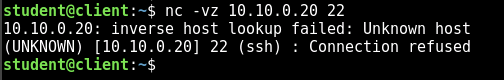
sudo iptables -A OUTPUT -m owner --uid-owner student -p tcp --dport 22 -d 10.10.0.20 -j REJECT

sudo iptables -L



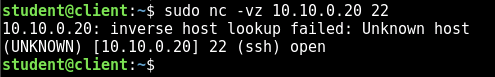
Now, let's try to connect again.

nc -vz 10.10.0.20 22



Let's try it again, running it as root.

sudo nc -vz 10.10.0.20 22



This time it worked and the connection was successful. As you can see, the connection is only blocked if it is attempted by a process running as **student**.

PreviousNext: Challenge

Live Chat

Securing Linux - Advanced IPTables

27 Minutes Remaining

Instructions Resources Help  100%

Challenge

Interfaces are allowed to have multiple IP addresses. These can be added with the following command.

sudo ip a add 192.168.1.150 dev eth1

Based on the content of this lab, and a little extra research on the **mark** module, see if you can determine how to select a custom source IP address for a packet based on the user that is running the process. That is, if user **student** is running Firefox, it will use the first IP address. If the user **root** is running Firefox, it will use the second IP address, and so on.

PreviousNext: Reference

Live Chat

Securing Linux - Advanced IPTables

27 Minutes Remaining

Instructions Resources Help  100%

Reference

1. Enable NAT

Apply NAT logic to packets outgoing on interface **eth0** and going to addresses other than **10.10.0.0/24**. Use the IP address **10.10.0.10** as the new address.

sudo iptables -t nat -A POSTROUTING ! -d 10.10.0.0/24 -o eth0 -j SNAT --to-source 10.10.0.10

1. Port Forwarding

Forward incoming packets on port 2222 to port 22 on 192.168.1.30.

sudo iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 2222 -j DNAT --to 192.168.1.30:22

1. Rate Limiting

Limit module:

sudo iptables -N limit-port-scanning

sudo iptables -A limit-port-scanning -p tcp --tcp-flags SYN,ACK,FIN,RST SYN -m limit --limit 1/s --limit-burst 2 -j RETURN

sudo iptables -A limit-port-scanning -p tcp --tcp-flags SYN,ACK,FIN,RST SYN -j DROP

sudo iptables -I INPUT 1 -j limit-port-scanning

Hashlimit module:

sudo iptables -A INPUT -p icmp -m hashlimit --hashlimit-name ICMPTEST --hashlimit-mode srcip --hashlimit-srcmask 32 --hashlimit-above 5/second --hashlimit-burst 2 --hashlimit-htable-expire 3000 -j DROP

Recent module:

sudo iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -m recent --set

sudo iptables -A INPUT -p tcp --dport ssh -m conntrack --ctstate NEW -m recent --update --seconds 20 --hitcount 10 -j DROP

Connlimit module:

sudo iptables -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 2 -j REJECT

1. Block Invalid TCP Flags
2. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,SYN,RST,PSH,ACK,URG NONE -j DROP
3. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,SYN FIN,SYN -j DROP
4. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags SYN,RST SYN,RST -j DROP
5. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,RST FIN,RST -j DROP
6. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags FIN,ACK FIN -j DROP
7. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,URG URG -j DROP
8. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,FIN FIN -j DROP
9. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ACK,PSH PSH -j DROP
10. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL ALL -j DROP
11. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL NONE -j DROP
12. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL FIN,PSH,URG -j DROP
13. sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL SYN,FIN,PSH,URG -j DROP

sudo iptables -t mangle -A PREROUTING -p tcp --tcp-flags ALL SYN,RST,ACK,FIN,URG -j DROP

1. Block Based on String Content

sudo iptables -A FORWARD -m string --string 'GET' --algo bm -j DROP

1. Block Based on User of Process

sudo iptables -A OUTPUT -m owner --uid-owner student -p tcp --dport 22 -d 10.10.0.20 -j REJECT

PreviousEnd

Live Chat