02233 Network Security
Assignments: Threat detection

27-02-2024

Today we will learn to configure firewall rules using `iptables`. In addition, we will learn how to setup a honeypot (Cowrie), interact, and analyze interactions. You can use your Kali VM to finish all the exercises, but you can always choose a different setup. Keep in mind that `iptables` is only available in Linux distros. For Cowrie in particular, we recommend using Docker, but you can install it from the source if you prefer.

Through the following firewall exercises, we will create rules that build towards more complex chains that will help you harden the system's security. Remember to flush your rules after each exercise.

1 Firewalls

If you remember the exercises from the previous week, we learned how to start a simple TCP server. Start the server to test your rules. As an alternative, you can use the following command to start a simple HTTP server:

```
python3 -m http.server 9000
```

Exercise 1: Blocking incoming and outgoing traffic. We will create two simple rules. The first should drop all the incoming traffic. For the second, create a rule to drop all outgoing traffic. Now test that your machine drops all the traffic. You can use Wireshark to verify that nothing goes through.

Solution

The recommended way of getting accustomed to `iptables` is reading the manpage, which you can do in your Kali terminal by running `man iptables`, or you can also look online, such as on die.net: `https://linux.die.net/man/8/iptables`.

The rule to drop all incoming traffic (line 1 below) is **A**ppended to the `INPUT` chain and instructs all matched packets (in this case, all of them, because we do not specify any matching rules) to **j**ump to the special builtin target `DROP`, discarding them. Dropping outgoing traffic is analogical, simply changing the source chain (line 2):

```
1 iptables -A INPUT -j DROP
2 iptables -A OUTPUT -j DROP
```

You will usually need to run `iptables` commands with root privileges. You can test that this works by trying to ping outside the VM, such as by running the `ping 8.8.8.8` command or trying to access a website through your browser, and inspecting the traffic in Wireshark. You should not be able to succeed in this communication.

Exercise 2: Blocking specific requests. Create a rule that blocks incoming HTTP traffic, and test it by trying to navigate to your web server. Then, create a rule that blocks ping incoming requests. The ping command uses the ICMP protocol with a header that specifies it wants a response (some sort of echo). Can you improve these rules to block flood attacks?

Hint: The idea behind a flood attack is to send ping requests to the target at a very high rate.

Solution

First, make sure that you clear the rules from the previous exercise by *flushing* `iptables` by running the `iptables -F` command.

A standard rule blocking incoming HTTP traffic can look like quite similar to the previous: **A**ppended to the `INPUT` chain, matching the `TCP` **p**rotocol on **d**estination **port** 80 (the standard port for HTTP): `iptables -A INPUT -p tcp --dport 80 -j DROP`

However, since our server is running on port 9000 rather than 80, traffic to that will **not** be blocked by this rule. You can either start the HTTP server on port 80 instead or specify the correct **d**estination **port** in the rule: `iptables -A INPUT -p tcp --dport 9000 -j DROP`

Incoming ICMP ping **requests** can be blocked using this rule:

```
iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

Note that if you do not specify the `echo-request` part, you will also block *outgoing* pings, since you will block **all ICMP communication**, including ICMP responses (pongs) from the hosts you yourself ping.

To block incoming ICMP request floods, you should first simulate one: use the `ping` command, setting the **i**nterval to 0.1 seconds: `ping -i 0.1 127.0.0.1`. This should work now. To prevent it, use the `limit` **m**odule, setting a limit of, for example, **1** ping per **s**econd (line 1). Rules using this module will match *until* the limit is reached, so you should `ACCEPT` these packets and `DROP` the rest, which can be done either by adding another rule *after* the limiting one (since packets that matched the first rule will not reach the second one; line), or by configuring a default `DROP` **P**olicy (line 5) for the chain.

```
1 iptables -A INPUT -p icmp --icmp-type echo-request -m limit
  ↪  --limit 1/s -j ACCEPT
2 # and
3 iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
4 # or
5 iptables -P INPUT DROP
```

(Bonus) Exercise 3.1: Advanced rules and chains. With iptables, we are capable of chaining rules using the `-N <name>` argument. This allows us to define groups of rules that will validate the traffic forwarded to the chain (see listing 1 for an example). For now, let's create a chain that restricts the use to our Virtual Machine (VM). The VM contains some services that need to remain accessible, but we don't need to expose the rest or allow outgoing connections. Write a simple chain that allows only our host to connect to port 22.

```
## Chain for a VM which only allows the same host to connect 3 times to the HTTP server every 24h.
# 1. Create the chain
iptables -N HTTP
# 2. Accept connections from a host up to 3 times, otherwise the connection attempt is rejected with reset.
iptables -A HTTP -p tcp --syn --dport 80  -m connlimit --connlimit-above 3 --connlimit-mask 32 -j REJECT --reject-with tcp-reset
# 3. Accept established connections from the same host that we have not seen in 24 hours for a maximum of three times.
iptables -A HTTP -p tcp --dport 80 -m state --state ESTABLISHED -m recent --name httpclient --rcheck --seconds 86400 --hitcount 3 -j ACCEPT
# 4. Log the rejected connections, so we can have
# a list of blocked IP addresses.
iptables -A HTTP -p tcp --dport 80 -m recent --name httpclient --set -j LOG --log-prefix "HTTP connection rejected: "
# 5. Forward any incoming traffic to the HTTP rule
iptables -A INPUT -p tcp --dport 80 -j HTTP
```

Listing 1: Example of iptables chain of rules to allow host connect a maximum of 3 times every 24h

Solution
You should clear the rules from the previous exercise by *flushing* `iptables` by running the `sudo iptables -F` and delete the chain by `sudo iptables -X SSH` (if chain *SSH* already exists).
Make sure you have the following general policies:

```
sudo iptables -P FORWARD DROP
sudo iptables -P OUTPUT ACCEPT
sudo iptables -P INPUT DROP
```

Next, you create the **SSH** chain and set up rules to manage SSH connections:

```
sudo iptables -N SSH
sudo iptables -A INPUT -p tcp --dport 22 -j SSH
sudo iptables -A SSH -p tcp -s <HOST_IP> --dport 22 -j
↪  ACCEPT
sudo iptables -A SSH -j DROP
```

For connections that have already been established, allow them to continue:

```
sudo iptables -A INPUT -m conntrack --ctstate
↪  ESTABLISHED,RELATED -j ACCEPT
```

Finally, to test if your SSH rule works as intended, attempt to SSH into your server from the specified host IP: `ssh root@<VM_IP> -p 22` Remember to replace `<HOST_IP>` and `<VM_IP>` with the actual IP addresses of your VM (see `ip addr`).

(Bonus) Exercise 3.2: Advanced rules and chains. Now write another chain to restrict access to the SSH service in Cowrie (port 2222). This chain should allow each host to connect to the service a maximum of 3 times every 24 hours. *If you want to challenge yourself even further, set a time limit of 60 seconds for each connection.*

Solution

Again, make sure that you clear the rules from the previous exercise by *flushing* `iptables` by running the commands below to remove all existing rules and custom chains. This ensures a clean state for setting up new rules.

```
# To clean up the rules from the previous assignments:
sudo iptables -F
sudo iptables -X
# Set default policies
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
```

After ensuring a clean slate, you proceed to create a new chain specifically for managing connections to the Cowrie SSH service, which runs on port 2222.

```
# Create the COWRIE chain
sudo iptables -N COWRIE

# Redirect incoming traffic on port 2222 to the COWRIE
↪  chain
sudo iptables -A INPUT -p tcp --dport 2222 -j COWRIE
```

To limit the number of connections each IP can make to the Cowrie SSH service to 3 per day, the following rules are added to the COWRIE chain:

```
# Allow up to 3 connections per IP per day
sudo iptables -A COWRIE -p tcp --dport 2222 -m state
↪  --state NEW -m recent --set --name cowrie
sudo iptables -A COWRIE -p tcp --dport 2222 -m state
↪  --state NEW -m recent --update --seconds 86400
↪  --hitcount 4 --name cowrie -j REJECT --reject-with
↪  tcp-reset
```

2 Honeypots - Cowrie

First, install and run Cowrie (see listing 2). One of the best features of Cowrie is the ability to replay logs. When you are done, navigate to your cowrie installation and check the newly created log file with our attack.

```
# Download and extract Cowrie
wget https://github.com/cowrie/cowrie/archive/refs/heads/master.zip
unzip master.zip
mv cowrie-master cowrie
# Create a virtual environment for Cowrie
cd cowrie
virtualenv cowrie-env
source cowrie-env/bin/activate
# Install Cowrie
pip install -U pip
pip install -r requirements.txt
cp etc/cowrie.cfg.dist etc/cowrie.cfg
# To start Cowrie run
bin/cowrie start
```

Listing 2: Cowrie installation

In this exercise, we will play with the SSH protocol in Cowrie and then we will replay our actions. For this, establish an SSH connection with the VM in port 2222 and perform some of the following *tasks (These tasks are typical procedures for installing a backdoor into a system).*

Solution
First, it is important to realize that now, you are playing the attacker role: you are trying to connect to a possibly vulnerable server and gain persistence by installing a backdoor, so you can connect to it even if the original vulnerability is closed). With that in mind, connect through SSH to Cowrie, which is running on `localhost` on **p**ort 2222. You should try logging in as the `root` user and try some easy-to-guess passwords (such as "root"): `ssh -p 2222 root@localhost`

1. Place your public SSH key in the `~/.ssh/authorized_keys` file. This should allow you to log in to the SSH server without the need for credentials. Typically, you will need to change the permissions of the file to make it readable only by the current user and restart the `sshd` service.

Solution
To do that, you should generate a keypair outside the SSH connection. You can use `ssh-keygen` to generate a key of **t**ype `RSA` with length of 2048 **b**its and store it in a **f**ile `my_ssh_key` in the local directory (.):

```
ssh-keygen -t rsa -b 2048 -f ./my_ssh_key
```

Afterwards, copy the contents of the `my_ssh_key.pub` file and connect through SSH. Try opening the `~/.ssh/authorized_keys` file there with your favorite ~~`vi`~~ text editor (you can also use `nano`). You will likely get an error such as this:

```
E558: Terminal entry not found in terminfo
```

Because you are not ready to give up just yet, you try using the `echo` command to append the key into the file, change the permissions and restart sshd:

```
echo "ssh-rsa ..." >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
service sshd restart
```

Disconnect (using ^D or the `exit` command) and try connecting again. The server will still ask you for a password, so enter the empty one again and check to see why this is the case:

```
cat ~/.ssh/authorized_keys
```

The file does not exist (because Cowrie does not persist these changes between sessions).

2. Try to download, install, and use some tool, e.g., Nmap.

Solution
The standard APT command should work and "install" nmap:

```
apt-get install nmap
```

However, many students had issues with this step – no output was presented. This is likely a networking issue and is not detrimental for the lab. It just serves to show a perhaps strange behavior of the remote terminal.
You can use `ping` instead to try pinging public IPs (such as 8.8.8.8), domains that exist (dtu.dk), domains that do not (this.doesnot.exist). You will always get a similar (successful) result (unlike in a real shell, where the nonexistent hosts would either not resolve or time out.

3. Did you notice anything strange?

Solution
We would think so :)
For example: the root password is empty, SSH keys are not taken into account, files are not persisted, packages are installed in a funky way, you can ping nonexistent hosts, the terminal in general behaves strangely...

4. Could you tell the difference between the honeypot and a real SSH service?

Solution
We run nmap using the following command:
`sudo nmap -sV -p 2222 localhost`

- -sV : service and version information
- -p PORT : specified port

By observing the nmap results, it looks like a legitimate running ssh service. We can also confirm the scan's interaction with the service by looking at `cowrie.log`.

2.1 Replaying the Logs

Solution
Cowrie records all sessions, so the administrator can later look at what the attacker was trying to do on the system (honeypot). You can list those recordings and replay them like this:

```
ls -lah var/lib/cowrie/tty
bin/playlog var/lib/cowrie/tty/fc7a392f0ddbe529be...
```