Secure Your Ssh Server with Iptables and IP Netblocks API

Posted on March 22, 2022

Secure shell (ssh) is the typical tool for getting secure command-line access to Linux (and other Unix flavor) systems. Notably, most Linux-based servers are administered remotely via ssh access. Hence the security of the ssh service is of paramount importance, especially since it is often a very attractive part of the attack surface of an organization.

The present blog provides a discussion on setting up efficient firewall rules for the ssh service, and extending the whitelist easily with the help of WhoisXML API's IP Netblocks API. The method also works for other services using inbound tcp connections. We discuss a typical iptables firewall on a Linux system. Basic expertise in Linux tools and firewalls is assumed. The recipe works as it is, or with minor modifications also on other systems.

1. Secure shell (ssh): attacks and mitigation measures

If an ssh service appears on a machine with a public IP, it will be surely attacked, no matter which port the ssh service is using. Dictionary attacks are the most common ones. They use a large dictionary of pwned accounts (i.e., whose username-password pair was stolen) to log in to the service.

As a preventive measure, it is certainly recommended to disable password-based authentication and use secure key pairs with complex passphrases. This decreases the likelihood of successful attacks. However, the appearance of failed login attempts in the access log is still annoying. The ssh keys can also be compromised. In addition, password-based authentication is needed in some cases. For instance, in a critical system it can be useful if the appropriate personnel can log in anytime with a complex enough password, without having an ssh private key at hand. Storing private keys on arbitrary devices is also a security risk. So it is better to have firewall rules.

The fail2ban tool is obviously a must-have: it will filter out IPs temporarily or eternally if it finds that there were failed login attempts. It works by scanning log files of services and adding firewall rules; ssh is the one configured by default in a basic installation. If a dictionary attack is conducted from a fixed IP address, it will efficiently filter it out. However, to detect an attack, it needs at least one failed attempt. In addition, distributed attacks are more common nowadays. The opponent will use a botnet or another kind of network of compromised machines for the login attempts. The auth.log during such an attack looks like this:

```
Feb 13 00:17:01 exampleserver sshd[19778]: Invalid user fangce from 49.235.
Feb 13 00:17:01 exampleserver sshd[19778]: Received disconnect from 49.235.
Feb 13 00:17:01 exampleserver sshd[19778]: Disconnected from invalid user f
Feb 13 00:23:35 exampleserver sshd[19797]: Received disconnect from 179.113
Feb 13 00:23:35 exampleserver sshd[19797]: Disconnected from authenticating
Feb 13 00:55:13 exampleserver sshd[19834]: Received disconnect from 106.53.
Feb 13 00:55:13 exampleserver sshd[19834]: Disconnected from authenticating
Feb 13 00:56:58 exampleserver sshd[19838]: Invalid user tec from 106.12.144
Feb 13 00:56:59 exampleserver sshd[19838]: Received disconnect from 106.12.
```

2. Protection against distributed dictionary attacks: considerations

Our goal is to prevent an unwanted IP address even from starting to authenticate with ssh. But what makes an IP address unwanted? For a critical system obviously we allow for a few fixed IPs only. But what if the ssh should be available to personnel using arbitrary Internet access like

cellular phones, possibly from multiple providers?

It is tempting to think of a solution that analyzes the IP address and lets it in if and only if the analysis concludes positively. One may, for instance, do an IP geolocation lookup for that purpose, and allow IPs from prescribed countries and regions. Such a solution is not easy to implement in a safe manner though. A complex analysis upon each connection makes our system prone to an evil form of denial-of-service attack. Namely, if our opponent knows about our solution, he can easily fork a tremendous amount of jobs on our system by initiating many connections. Even though the dictionary attack we want to fight is small by nature, we do not want to introduce another vulnerability when fighting it.

So we are left with the idea of defining firewall rules in advance. Certainly by default we will not allow any inbound connections on the port of ssh. We can set up a set of whitelisted IP addresses, too. But what if we want to allow more?

The obvious idea is to let IPs in from selected networks. In what follows we consider IPv4 addresses. The IPv4 space is subdivided into a hierarchy of contiguous ranges termed as netblocks. Given an IP, how do we find out what is the range to whitelist in order to allow all connections, e.g., from an ISP's, company's or a university's network?

An IP WHOIS query of an IP address results in a list of blocks the IP belongs to via an IP Whois lookup. Our solution will be:

- We will have a set of iptables rules to let inbound tcp connections to ssh's port only from a whitelist of IP addresses and netblocks.

- We will implement a utility to find out the netblocks an IP address belongs to, and let us add extra networks to the whitelist based on this information.

3. Netblock-based whitelisting: implementation

Our example implementation can be found on GitHub at: https://github.com/whois-api-llc/whoisapi_tcp_netblocks_whitelist

Here we discuss its principles in more detail.

3.1. Updating firewall rules

As of the script to maintain the firewall rules: we use the INPUT chain of iptables by default and the iptables command to maintain it. Our script implements the following algorithm:

- Read the list of whitelisted IPs and ranges (in CIDR notation, like 192.168.0.0/16) notation from a whitelist file. The file has one IP or CIDR each line.
- List the already existing rules for inbound tcp connections to ssh's port.
- Compare the lists. If there is a new element to whitelist in the file, insert it to the beginning of the chain with ACCEPT.
- Check if the default generic rule for inbound tcp connections to ssh's port is REJECT; were this not the case, append this rule at the end of the chain.

An example implementation of this is the firewall_tcp_update_iptables.sh script in the GitHub project. It has no arguments and can be run repeatedly anytime.

3.2. Adding new ranges comfortably

To add ranges to the whitelist, one can of course add particular IP addresses and netblocks in CIDR notation manually. In the case of netblocks, however, it takes time to find out what the right netblocks are. IP WHOIS data can be queried directly, e.g., with the whois command, however, the result is unstructured, and thus it takes some programming effort to parse it.

This is where the IP netblocks API comes into play. Given an IP address, it will serve the ownership and data of the netblocks the address belongs to in a handy JSON format. An API key is required to use it, which can be obtained freely after registration at the API website. Extensive lookups need a paid subscription, but the free version will be sufficient for the present use case in most cases. We implement the script to maintain our whitelist file in Python. For the IP netblocks

API, a Python library is also offered that can be installed simply with Python's package manager:

```
pip install ip-netblocks
```

Getting IP Whois data in a handy Python data structure is then as simple as

```
from ipnetblocks import *
client = Client('Your API key')
response = client.get('8.8.8.8')
```

(The library can do even much more; consult its documentation).

Armed with these tools, we can now easily implement the following algorithm: The input is an IPv4 address.

- Get the netblocks the input address belongs to,
- Give information to the user interactively, and let the user choose the right block,
- Convert the chosen netblock to contiguous CIDR blocks with Python's netaddr package~,
- Read the whitelist file,
- Add those CIDR blocks which are not yet there,
- Overwrite the whitelist file with the new list.

The implemented script, firewall_tcp_add_ips_net_to_whitelist.py in the GitHub package works as follows: assuming e.g. you want to add a netblock which 8.8.8.8 belongs to, run it like this:

```
./firewall_tcp_add_ips_net_to_whitelist.py 8.8.8.8
```

The session will read:

```
1 8.8.8.0 - 8.8.8.255 LVLT-GOGL-8-8-8-[]
2 8.0.0.0 - 8.15.255.255 LVLT-ORG-8-8-[]
3 8.0.0.0 - 8.127.255.255 LVLT-ORG-8-8-[]
4 8.0.0.0 - 8.255.255.255 NET8-[]
5 6.0.0.0 - 13.115.255.255 NON-RIPE-NCC-MANAGED-ADDRESS-BLOCK-['IPv4 addres
6 0.0.0.0 - 255.255.255.255 IANA-IPV4-MAPPED-ADDRESS-[]
7 :: - ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff IANA-BLK-['The whole IPv6 ad
Which one you choose? (Default: 1) 1
Add 8.8.8.0 - 8.8.8.255 LVLT-GOGL-8-8-8-[] (y/n)?
```

The whitelist file will now contain 8.8.8.0/24. (Caution: if you're testing it from an ssh connection, it is better to add your current IP, or maybe its range to the whitelist file manually at the beginning, to avoid locking yourself out from the system.) Then run firewall_tcp_update_iptables.sh to activate your new settings. Instead on 8.8.8.8 you will probably want to add, e.g., a range belonging to your ISP or phone network provider where your phone gets your IPs from.

Conclusions

The described solution is a simple way to get rid of many invalid attempts to log in to an ssh server. It can also be used to protect other kinds of services; for those using inbound tcp connections to a port, even without any modification. With the help of the IP netblocks API it is easy to extend the list of the allowed networks interactively. Certainly the idea, and also the implementation can be extended in many different ways.