SSH reverse shells

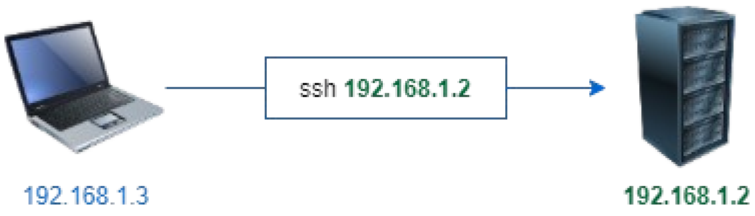
Maxime Moreillon

Nov 21, 2021

SSH is one of the most widely used network protocol for interacting with remote servers and computers. However, in some cases, SSH access to a specific host can be blocked due to firewalls or other network settings. This article presents how an SSH reverse shell could help circumvent the problem.

The situation

Let us begin with a simple scenario where we want to SSH into a given server. For this example, the IP of this server is 192.168.1.2 while that our computer is 192.168.1.3.

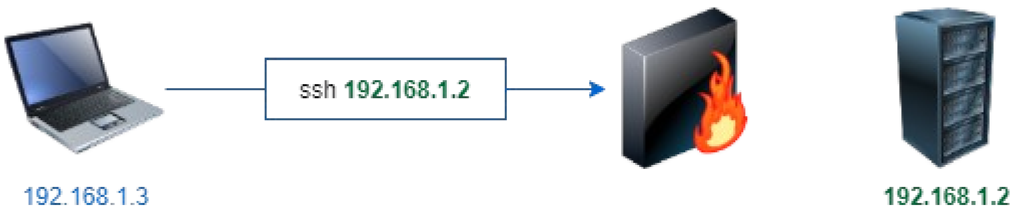


In the absence of any constraint, this can be achieved trivially with the basic SSH command:

```
ssh 192.168.1.2
```

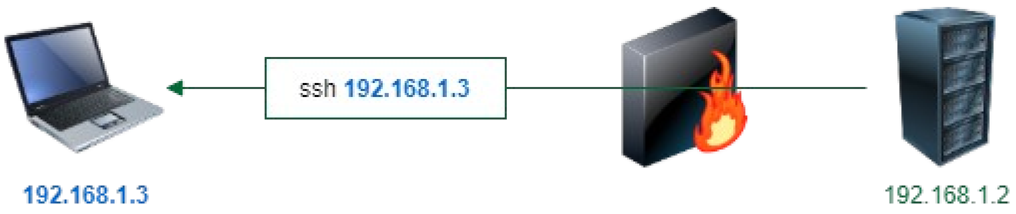
Note here that the username has been omitted. To specify a username, simply prefix it to the address followed by a @ (for instance user@192.168.1.2).

Now if, for example, 192.168.1.2 sits behind a firewall that blocks inbound traffic on all ports, SSH connection attempts from 192.168.1.3 will be blocked



Reverse shells to the rescue

Firewall are often configured asymmetrically, blocking traffic in one direction but not the other. Thus, it would not be unnatural that 192.168.1.2 can connect to other hosts via SSH even itself is to reachable.



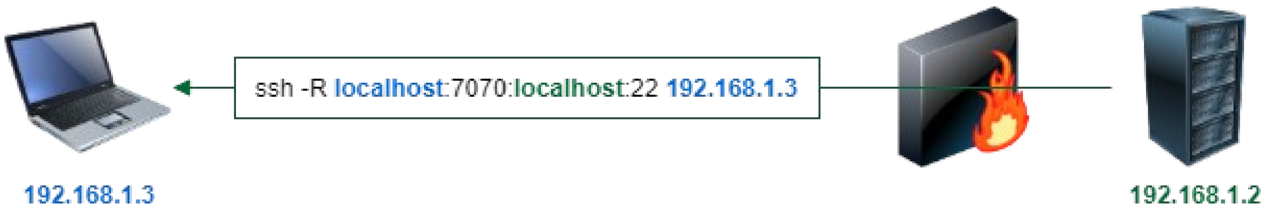
As such, an SSH connection can existing between the two hosts, although not in the direction that we would want. But what if 192.168.1.3 could use such a connection in reverse direction? This is the core idea behind a reverse shell.

Although not possible by default, an SSH connection can actually be configured to allow reverse connections using the -R flag.

The parameters of the -R flag take the following form: *<host on destination>:<port on destination>:<host on origin>:<port on origin>*.

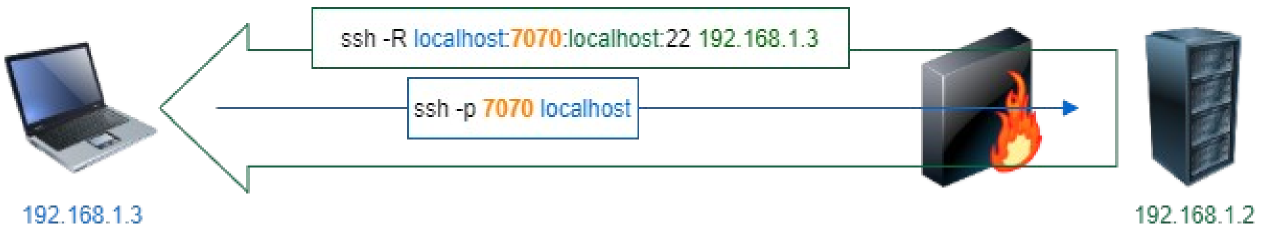
For the time being, we want an SSH connection into 192.168.1.2 itself so *<port on origin>* is set to 22 while *<host on origin>* can just be set to localhost. On the other hand we want our computer to be able to connect to 192.168.1.2 using a port of our choice, *<port on destination>*. Here 7070 is used as an example and is thus opened on 192.168.1.3 itself. Our computer will use port 7070 opened on itself to access 192.168.1.2 and thus *<host on destination>* is set to localhost. As this all sounds confusing, here is how the command would look for this example:

```
ssh -R localhost:7070:localhost:22 192.168.1.3
```



With such connection established, 192.168.1.3 can use the chosen port, i.e. 7070, to connect to 192.168.1.2.

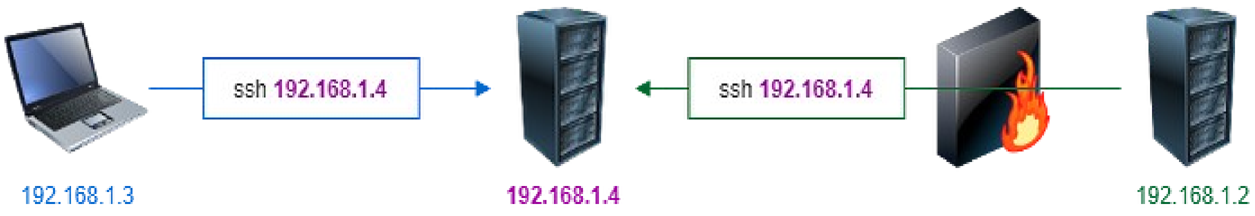
```
ssh -p 7070 localhost
```



Using a gateway server

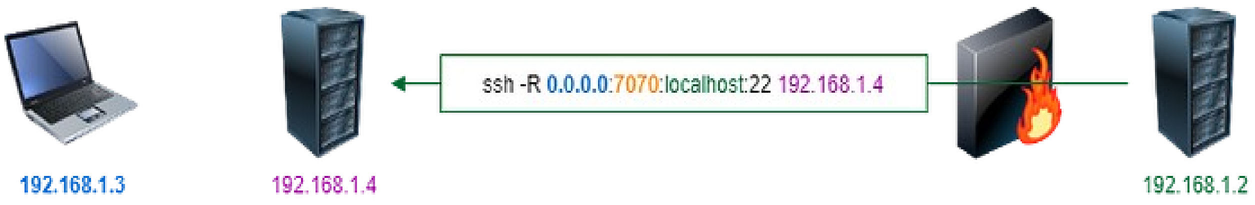
With this approach, an SSH connection from our computer, 192.168.1.3, to our target, 192.168.1.2, can be established. However, if our computer is a device like a laptop, it might not be turned on or available all the time. This quickly becomes a problem since our connection to the target first requires the latter to first connect to our machine. As such, one would need to access the target to start a new connection each one the previous one got terminated and if such access was possible, then a reverse shell would not have been necessary in the first place.

Instead, we would prefer to use one server as gateway to the target, which our laptop can easily connect into whenever access to 192.168.1.2 is required. Let's imagine that this gateway server exists at the address 192.168.1.4 and is accessible by both our machine and our target.



We want to allocate port 7070 of the gateway so that computers like ours can use it to SSH into 192.168.1.2. To achieve this, we SSH from 192.168.1.2 into 192.168.1.4 using the -R flag but this time, the *host on destination* option mentioned earlier is set to 0.0.0.0. This implies that the destination allows any host to connect on the given port, here 7070:

```
ssh -R 0.0.0.0:7070:localhost:22 192.168.1.4
```

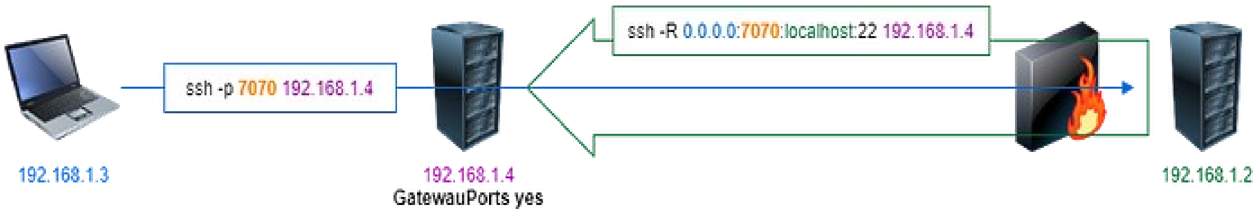


Now, in order to allow 192.168.1.3 (a host external to 192.168.1.4) to use port 7070 for connection, the gateway's SSH server needs to be configured appropriately. This is done by editing the /etc/ssh/sshd_config file and removing the comment at the line that says *GatewayPorts yes:*

```
...
#AllowTcpForwarding yes
GatewayPorts yes
X11Forwarding yes
...
```

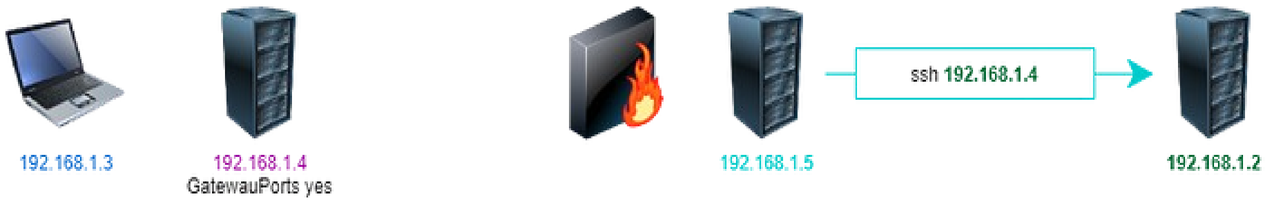
With this done, our machine can now use the connection between the target and the gateway in reverse using the following:

```
ssh -p 7070 192.168.1.4
```

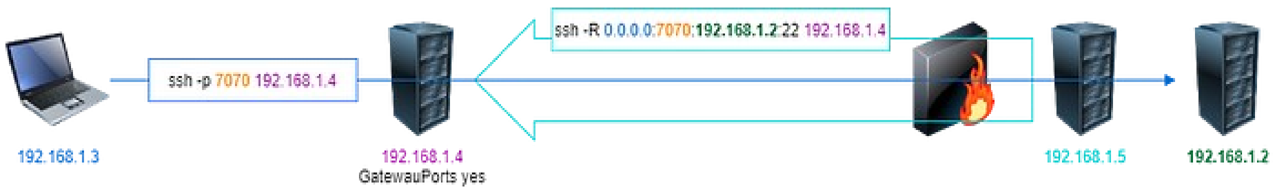


Connecting to another target server

In some cases, the server we want to connect to - in this case, 192.168.1.2 - might not be suitable to hold the reverse SSH connection. This issue can be solved if another server which can establish a direct connection is available for the task. Here, let us assume that this server has the IP address 192.168.1.5.



Here we can use 192.168.1.5 as relay for our SSH connection. For that purpose, 192.168.1.5 opens the SSH connection to our gateway with the -R flag, but this time specifying 192.168.1.2 as the destination instead of localhost. On our side, nothing changes and we can still access the target by connecting to port 7070 of the gateway.



A SystemD service to manage the connection

We now have an SSH connection established from our target to our gateway server, which we can use in reverse to interact with the former. However, if for some reason this connection gets broken, we would need to get access to the target to restart it. It would be much preferable if this connection existed as a service that automatically restarts whenever it fails.

To achieve this, we can write a simple service file for SystemD and just enable it for automatic start on boot.

For example, one can create a file called reverse_shell.service in /etc/systemd/system with the following content:

```
[Unit]
Description="Reverse SSH"
After=network.target[Service]
User=YOUR_USER
Restart=always
RestartSec=10sExecStart=/usr/bin/sshpass -p "YOUR_PASSWORD" \
    ssh -N \
    -R 0.0.0.0:7070:localhost:22 \
    -o ExitOnForwardFailure=yes \
    -o ProtocolKeepAlives=5 \
    192.168.1.4[Install]
WantedBy=multi-user.target
```

A few things to note about this service file:

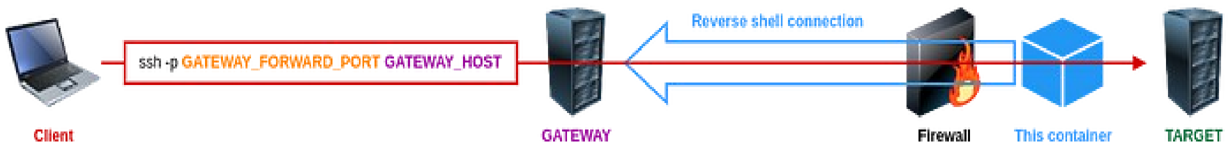
- The password needed to connect to 192.168.1.4 is injected using sshpass and having it in plain text like that is very insecure
- The SSH connections is made non-interactive with the -N flag
- The connection is kept alive using the -o ProtocolKeepAlives=5 flag

Such file can then be enabled with

```
sudo systemctl enable reverse_shell.service
```

Reverse SSH in a docker container

The method presented in this article simply requires having a host server that can SSH into both the target and the gateway. The nature of such host is simply limited by whether it can run the SSH command with the -R flag. As such, it does not necessarily need to be a fully fledged server or desktop computer. With such degree of freedom in mind, I designed a Docker container which handles the reverse SSH mechanism presented in this article.



The docker container image is available on Docker Hub and can be run easily, specifying connection parameters as environment variables. For the example presented in this article, the command would look like:

```
docker run \
-e GATEWAY_HOST=192.168.1.4 \
-e GATEWAY_FORWARD_PORT=7070 \
-e GATEWAY_USERNAME=GATEWWAY_USERNAME \
-e GATEWAY_PASSWORD=GATEWWAY_PASSWORD \
-e TARGET_HOST=172.16.1.2 \
moreillon/reverse-ssh
```

More information about this container on the project's repository on GitLab.

Conclusion

Reverse shells are great ways to get remote access to computers and servers that would otherwise be unreachable due to network settings or firewalls. Obviously setting up such mechanism involves having access to the target machine at some point in time, but once the connection is in place, it can be left unattended.