< all posts

Building an SSH Jumpbox with Docker

Posted by Rob Beal on March 01, 2019 · 5 min read open source docker security

As part of our never-ending pursuit of staying secure, we have recently built an SSH jumpbox as a central, secure way to access our production instances on AWS. A fairly standard affair, although in this instance we solved the problem using Docker, numerous services (such as rsyslog and fail2ban) and related the jumpbox users to our AWS users for seamless management… so we thought we'd share how we did it!

Firstly... what is an SSH Jumpbox?

A jumpbox is a host you connect/tunnel through to access a target (hidden) host. It performs no additional function beyond helping create a secure tunnel between the user and the target (hidden) host. SSH is the underlying technology used between the user and jumpbox to form the tunnel. Using SSH any port can be mapped back to the user, in order to do so we need to allow the jumpbox to talk to that host via the port in question using an EC2 Security Group.

Why an SSH Jumpbox?

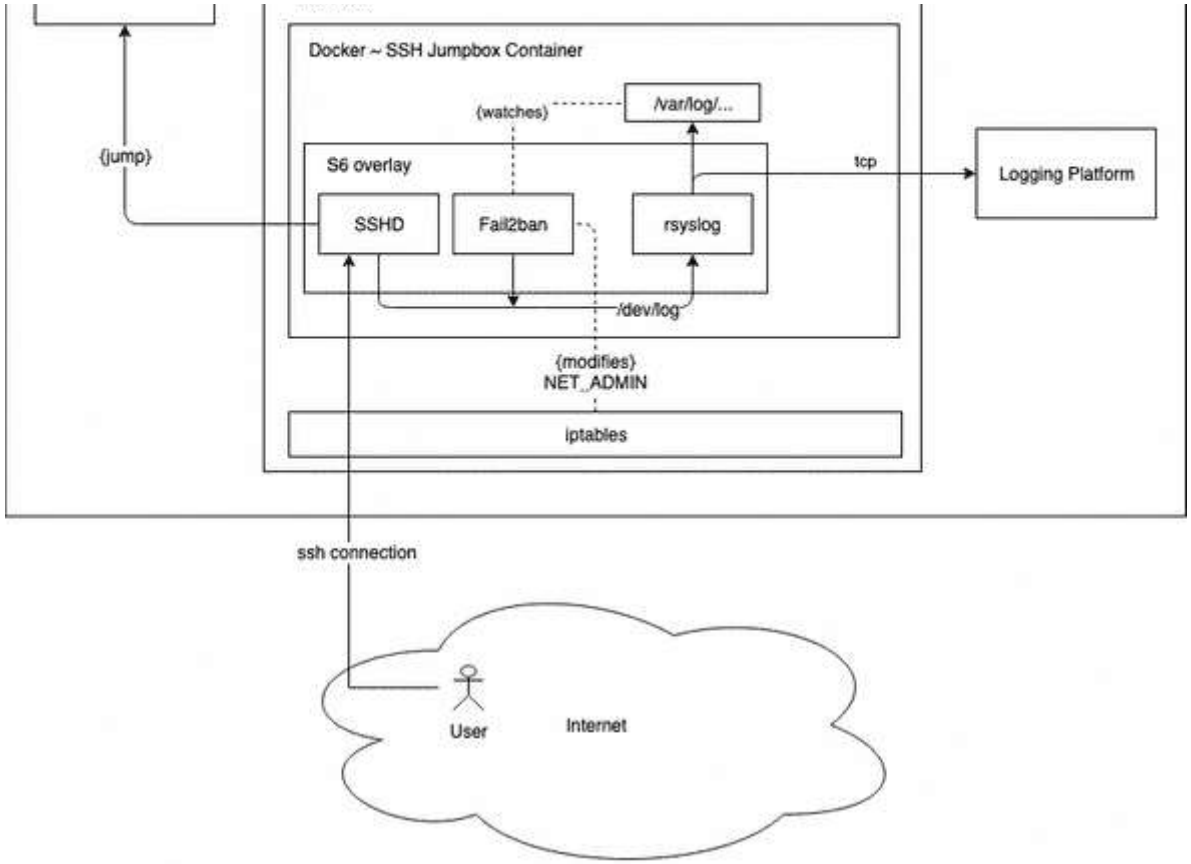
- **Simplicity -** we don't yet need the overhead of a VPN, a Jumpbox is enough for our current needs and far cheaper
- **Reduced Attack Surface -** fewer hosts are publicly exposed. Only the jumpbox is publicly accessible
- **Auditing** - logging access is simpler as all users access internal hosts via the jumpbox
- **Management -** we have a single public host to secure/maintain/update instead of numerous hosts
- **Single Responsibility -** the jumpbox performs a single function and performs it well

Building an SSH Jumpbox

We settled on using Docker for creating our jumpbox, hosted on an EC2 instance via ECS. Docker was largely chosen because of the fast feedback loop from being able to test-drive the container (via `testinfra`) to running/debugging it locally (and consistently) and tearing it up/down on AWS easily.

The container is based on an alpine image for a smaller size and attack surface, running a handful of processes:

- `openssh` for our SSH server. We have locked down `/etc/ssh/sshd_config`:
  - not allowing interactive mode (there's no reason to be on the jumpbox)
  - not allowing password auth (as it is less secure than key-based)
  - not allowing root login (as there should be no need to login as root)
  - forcing SSH protocol 2
- `rsyslog` for managing our logs (and sending them to our logging platform)
- `fail2ban` for banning malicious activity per ip address
- `s6` overlay as our process supervisor, managing all the above processes



Here is our `Dockerfile`:

```
FROM alpine:latest

ARG AWS_ACCESS_KEY_ID
ARG AWS_SECRET_ACCESS_KEY
ARG IGNORED_IPS
ARG LOGGING_DEST
ARG OVERLAY_VERSION=1.21.7.0
```

```
COPY create-users.sh .
COPY keys /etc/ssh
COPY s6 /etc/services.d

RUN apk add --no-cache fail2ban openssh openssh-server-pam grep rsyslog rsy
  && apk add --no-cache --repository http://uk.alpinelinux.org/alpine/edge/
  # s6 overlay
  && curl -L "https://github.com/just-containers/s6-overlay/releases/downlc
  # fail2ban
  && mv /etc/services.d/fail2ban/*.local /etc/fail2ban/ \
  && sed -i -e "s/{IGNORED_IPS}/$IGNORED_IPS/" /etc/fail2ban/jail.local \
  # logging
  && sed -i -e "s/{LOGGIN_DEST}/$LOGGING_DEST/" /etc/services.d/rsyslog/rsy
  # create users via aws
  && export AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID} \
  && export AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY} \
  && ./create-users.sh && rm create-users.sh \
  && chmod -R 600 /etc/ssh \
  && apk del --purge dependencies

EXPOSE 22

ENTRYPOINT ["/init"]
```

As part of building our jumpbox, we create a (password disabled) user on the jumpbox for each user on AWS. This is easily done using a bash script and `boto`. For each user created on the jumpbox, we get the public SSH key associated with respective AWS user and add it as an `~/.ssh/authorized_keys` (so the user is allowed to connect via SSH). This relates our AWS

users to the jumpbox users and means there is no user/key sharing happening and we have cleaner/clearer auditing as a consequence! If someone new starts or leaves, we simply need to update our AWS users and kick off a CI build and deploy (which takes no more than 2 minutes) to refresh the container.

```
#!/bin/bash

users=$(aws iam get-group --group-name users | jq '.Users[].UserName' -r) |

if [ -z "$users" ]; then
  echo "No users retrieved, please check your AWS credentials and access"
  exit 1
fi

while read -r username; do
  echo "Creating user '$username'"
  adduser -D "$username" || exit 1

  echo "Fetching ssh key ids..."
  ssh_key_ids=$(aws iam list-ssh-public-keys --user-name "$username" | jq '
  if [ -z "$ssh_key_ids" ]; then
    echo "No key ids found"
    continue
  fi

  echo "$ssh_key_ids"
  mkdir -p /home/"$username"/.ssh
```

```
    echo "Fetching ssh keys..."
    while read -r ssh_key_id; do
      ssh_key=$(aws iam get-ssh-public-key --user-name "$username" --ssh-publ
      echo "$ssh_key" >> /home/"$username"/.ssh/authorized_keys
      echo "$ssh_key"
    done <<< "$ssh_key_ids"

    chown -R "$username:$username" /home/"$username"/.ssh
    chmod 700 /home/"$username"/.ssh
    chmod 600 /home/"$username"/.ssh/authorized_keys
  done <<< "$users"
```

Testing our container

We use the rather awesome `testinfra` python package to help us test-drive our container. Using it we can test numerous things from packages installed to more complex tests such as checking a logging platform connection is 'ESTABLISHED' via `netstat`. We have over 20 tests, here's a snippet of some:

```
import boto
import os
import pytest
import subprocess
import testinfra
```

```
@pytest.fixture(scope='session')
def host(request):
    subprocess.check_call(['make', 'build'])
    docker_id = subprocess.check_output(['make', '--silent', 'daemonise']).

    yield testinfra.get_host("docker://" + docker_id)

    subprocess.check_call(['docker', 'rm', '-f', docker_id])

def test_sshd_process_is_running(host):
    process = host.process.get(comm='sshd')
    assert process.user == 'root'
    assert process.group == 'root'

def test_rsyslog_is_connected_to_logging_platform(host):
    port = os.environ['LOGGING_PLATFORM_PORT']
    assert host.run(f'netstat -atn | grep -P ":{port}\s+ESTABLISHED"').rc =

def test_users(host):
    iam = boto3.resource('iam', aws_access_key_id=os.environ['AWS_ACCESS_KE
    for user in iam.Group('users').users.all():
        assert host.user(user.name).exists

        home = '/home/%s' %user.name
        if host.file(home + '.ssh').exists:
            assert host.file(home + '.ssh').mode == 0o700

            authorized_keys = home + '/authorized_keys'
            assert host.file(authorized_keys).exists
```

```
assert host.file(authorized_keys).user == user.name
assert host.file(authorized_keys).mode == 0o600
```

Running our container

As `fail2ban` uses iptables under the hood for banning ip addresses, the container needs to run with slightly elevated privileges. This translates into giving the container `NET_ADMIN` privileges (this is an inbuilt Linux privilege), the minimum privileges needed in order for the container to be able to modify `iptables` of the host. In a `docker run` command this translates as `--cap-add=NET_ADMIN`.

The container can also run in `--readonly` mode meaning it can't be modified, just for that added bit of security. We mount our log directory into the container meaning we also retain our log files after a deploy.

open source docker security

We are the engineers behind LandInsight and LandEnhance. We're helping property professionals build more houses, one line of code at a time. We're based in London, and yes, we're hiring!

← Building our First Mobile App with React Native

What Have I Learnt??? →