# Gathering Entropy in the Cloud: Alpine Linux, rng-tools, and Kubernetes DaemonSet

Failing TLS connections! Long delays generating transaction ID GUIDs! We recently ran into an issue while working with a customer to deploy and operate containers in Azure Kubernetes Service (AKS). The AKS cluster worker nodes, based on Ubuntu 18.04, did not have an entropy gathering daemon running. Reads from /dev/random were blocking occasionally which impacted TLS negotiation and other cryptographic operations whenever random numbers were needed.

A fact well known among Linux system administrators is that Linux systems before kernel 5.6 require an entropy gathering daemon (EGD). Without an EGD, reads from /dev/random may block when the kernel does not have enough internal entropy to provide cryptographically secure random numbers. An EGD opens /dev/random and interacts with the kernel through special ioctl() commands on the file descriptor (see [random(4) random, urandom - kernel random number source devices](https://man7.org/linux/man-pages/man4/random.4.html) and [random(7) random - overview of interfaces for obtaining randomness](https://man7.org/linux/man-pages/man7/random.7.html) for more information).

This post describes how to run an entropy gathering daemon across all Linux Kubernetes worker nodes using:

* [Alpine Linux](https://alpinelinux.org/) minimal base image
* Docker image based on Alpine Linux with [rng-tools](https://github.com/nhorman/rng-tools) installed
* Kubernetes DaemonSet configuration to run the image as a daemon on all Linux worker nodes

We will show how to run an EGD based on rng-tools using three approaches:

1. **Temporary Approaches:** Quickly deploy a DaemonSet based on either an image using the haveged EGD or one that uses the alpine:latest base image with the commands to install rng-tools and then run rngd. Use these steps to quickly relieve a cluster that is failing due to blocking reads from /dev/random until a better approach can be implemented.
2. **Simple Image:** Create a container image like the temporary approach and then run a DaemonSet using that image. Most enterprise customers use their own container registry were containers are scanned and vetted before being deployed into clusters. This creates a simple initial image that will be optimized for size and reduced attack surface in the next step.
3. **Advanced Minimal Image:** Create a container image from scratch with just the necessary packages to run rngd using a multi-stage Docker build that reduces both image size and attack surface.

Before we dive into the details of these approaches, let's first see how to test performance of reading from /dev/random.

# Testing /dev/random Read Performance

It's important to test when making performance-related changes. We can easily test read performance by running a temporary pod on one of the worker nodes (assuming all worker nodes are Linux for the moment). To do this we use the kubectl run command to create a temporary pod. The pod executes a timed dd command to read 1MB from /dev/random. We also emit the level of entropy when the command completes.

| $ kubectl run entropy-test --image=alpine --image-pull-policy=IfNotPresent --rm --stdin --tty --wait -- sh -c 'time dd if=/dev/random of=/dev/null bs=1 count=1M; cat /proc/sys/kernel/random/entropy\_avail'  If you don't see a command prompt, try pressing enter.  1048576+0 records in  1048576+0 records out  real 0m 2.91s  user 0m 0.55s  sys 0m 2.31s  1059  Session ended, resume using 'kubectl attach entropy-test -c entropy-test -i -t' command when the pod is running  pod "entropy-test" deleted |
| --- |

The total time to read 1MB is 2.91 seconds or about 352KB/s throughput. When the command completed the entropy level was 1059 whereas the maximum entropy is 4096. If the system was very low on entropy then the dd command may execute for several seconds or minutes and the entropy level may be very low. Entropy levels less than 1000 are signs of low entropy condition.

# Temporary Approach

This approach is a simple temporary solution to quickly deploy an entropy gathering daemon into a cluster without the need of a custom container image and associated container registry. Containers should run based on a specific image but this workaround can be used when a Kubernetes cluster has no EGD at all and is floundering due to long waits blocked on /dev/random reads. This technique can be used to quickly recover a cluster from a low entropy state until better mitigations are assembled and deployed.

## Haveged DaemonSet

The first temporary approach deploys a pre-built image hortonworks/haveged:1.2.0 from Docker Hub.

**alpine-haveged-workaround.yaml**

| apiVersion: apps/v1  kind: DaemonSet  metadata:  name: haveged  labels:  app.kubernetes.io/name: haveged  spec:  selector:  matchLabels:  app.kubernetes.io/name: haveged  template:  metadata:  labels:  app.kubernetes.io/name: haveged  spec:  affinity:  nodeAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  nodeSelectorTerms:  **-** matchExpressions:  **-** key: kubernetes.io/os  operator: In  values:  **-** linux  containers:  **-** name: haveged  image: hortonworks/haveged:1.2.0  securityContext:  capabilities:  add: ["SYS\_ADMIN"] |
| --- |

We can then deploy it and check the haveged logs.

| $ kubectl apply -n default -f alpine-haveged-workaround.yaml  daemonset.apps/haveged created    $ kubectl logs -n default -l app.kubernetes.io/name=haveged  haveged: listening socket at 3  haveged starting up  haveged: listening socket at 3  haveged starting up  haveged: listening socket at 3  haveged starting up  haveged: listening socket at 3  haveged starting up |
| --- |

The daemon looks operational. Let's delete it and move onto the other temporary approach.

| $ kubectl delete -n default daemonsets.apps haveged  daemonset.apps "haveged" deleted |
| --- |

## Alpine Linux and rng-tools DaemonSet

We'll create a simple Kubernetes [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) to run a pod on each worker node. This DaemonSet YAML configuration will form the basis of later steps and contains a couple of special configurations:

1. The command only needs to run on Kubernetes worker nodes that are running Linux. We set nodeAffinity to check that the kubernetes.io/os annotation is set to linux.
2. We need to ensure that the pod runs with suitable permissions to deliver entropy to the kernel. The pod is configured to run within a securityContext with SYS\_ADMIN capability enabled.

Below is the alpine-rng-tools-workaround.yaml file that implements this approach. The core configuration run the command apk update && apk upgrade && apk add rng-tools && rngd --foreground to update Alpine Linux packages, upgrade the system to apply any security updates, add the rng-tools package, and finally execute the rngd daemon in the foreground.

**alpine-rng-tools-workaround.yaml**

| apiVersion: apps/v1  kind: DaemonSet  metadata:  name: rngd  labels:  app.kubernetes.io/name: rngd  spec:  selector:  matchLabels:  app.kubernetes.io/name: rngd  template:  metadata:  labels:  app.kubernetes.io/name: rngd  spec:  affinity:  nodeAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  nodeSelectorTerms:  **-** matchExpressions:  **-** key: kubernetes.io/os  operator: In  values:  **-** linux  containers:  **-** name: rngd  image: alpine:latest  command:  **-** sh  **-** -xc  **-** apk update && apk upgrade && apk add rng-tools && rngd --foreground  securityContext:  capabilities:  add: ["SYS\_ADMIN"] |
| --- |

We can now apply into our Kubernetes cluster and check that the daemon is running:

| $ kubectl apply -n default -f alpine-rng-tools-workaround.yaml  daemonset.apps/rngd created    $ kubectl get -n default daemonsets.apps  NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE  rngd 4 4 4 4 4 <none> 4m25s    $ kubectl logs -n default -l app.kubernetes.io/name=rngd --tail=99  + apk update  fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/main/x86\_64/APKINDEX.tar.gz  fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/community/x86\_64/APKINDEX.tar.gz  v3.14.2-119-g9c4e1aa60c [https://dl-cdn.alpinelinux.org/alpine/v3.14/main]  v3.14.2-120-g90167408c8 [https://dl-cdn.alpinelinux.org/alpine/v3.14/community]  OK: 14943 distinct packages available  + apk upgrade  (1/1) Upgrading alpine-keys (2.3-r1 -> 2.4-r0)  OK: 6 MiB in 14 packages  + apk add rng-tools  (1/2) Installing jitterentropy-library (3.0.1-r0)  (2/2) Installing rng-tools (6.12-r0)  Executing busybox-1.33.1-r3.trigger  OK: 6 MiB in 16 packages  + rngd --foreground  Initializing available sources  [hwrng ]: Initialization Failed  [rdrand]: Enabling RDSEED rng support  [rdrand]: Initialized  [jitter]: Initializing AES buffer  [jitter]: Enabling JITTER rng support  [jitter]: Initialized  unable to adjust write\_wakeup\_threshold: Read-only file system  ... |
| --- |

We see that the DaemonSet was created and the logs (from one instance) show the apk commands completed successfully with the final rngd daemon up and running.

It's generally frowned upon to run a container using only a base image and then installing and running software. Instead we will generate a custom image, upload it to a container registry, and then configure the DaemonSet to use that image.

# Simple Image Approach

We will create a new container image based on Alpine Linux, with the rng-tools package installed, and configured to run rngd. We will also remove the apk cache files to reduce the size of the image a bit.

**Dockerfile.simple**

| FROM alpine:latest  USER root  RUN set -eux; \  apk update \  && apk upgrade \  && apk add rng-tools \  && rm -rvf /var/cache/apk/\*  CMD ["rngd", "--foreground"] |
| --- |

Next, we just need to build the image, tag it for our remote container registry, and then push it to the remote registry (I re-pushed so one layer already existed).

| $ docker build -t alpine-rng-tools-simple:1.0 -f Dockerfile.simple .  Sending build context to Docker daemon 3.072kB  Step 1/4 : FROM alpine:latest  ---> 14119a10abf4  Step 2/4 : USER root  ---> Running in 836bfaf6d5d9  Removing intermediate container 836bfaf6d5d9  ---> cc44cddc7d00  Step 3/4 : RUN set -eux; apk update && apk upgrade && apk add rng-tools && rm -rvf /var/cache/apk/\*  ---> Running in f7a1756b0ee8  + apk update  fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/main/x86\_64/APKINDEX.tar.gz  fetch https://dl-cdn.alpinelinux.org/alpine/v3.14/community/x86\_64/APKINDEX.tar.gz  v3.14.2-119-g9c4e1aa60c [https://dl-cdn.alpinelinux.org/alpine/v3.14/main]  v3.14.2-120-g90167408c8 [https://dl-cdn.alpinelinux.org/alpine/v3.14/community]  OK: 14943 distinct packages available  + apk upgrade  (1/1) Upgrading alpine-keys (2.3-r1 -> 2.4-r0)  OK: 6 MiB in 14 packages  + apk add rng-tools  (1/2) Installing jitterentropy-library (3.0.1-r0)  (2/2) Installing rng-tools (6.12-r0)  Executing busybox-1.33.1-r3.trigger  OK: 6 MiB in 16 packages  + rm -rvf /var/cache/apk/APKINDEX.406b1341.tar.gz /var/cache/apk/APKINDEX.a251b1f2.tar.gz  removed '/var/cache/apk/APKINDEX.406b1341.tar.gz'  removed '/var/cache/apk/APKINDEX.a251b1f2.tar.gz'  Removing intermediate container f7a1756b0ee8  ---> a834961626dc  Step 4/4 : CMD ["rngd", "--foreground"]  ---> Running in 10c24af1712b  Removing intermediate container 10c24af1712b  ---> c2626da03c5c  Successfully built c2626da03c5c  Successfully tagged alpine-rng-tools-simple:1.0    $ docker tag alpine-rng-tools-simple:1.0 us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-simple:1.0    $ docker push us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-simple:1.0  The push refers to repository [us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-simple]  75d948c90c30: Pushed  e2eb06d8af82: Layer already exists  1.0: digest: sha256:63dc691663fd3880c6d963fb71279a347ede5baaa7b3342e626419cc50ad595c size: 737 |
| --- |

Next, we can update the DaemonSet configuration to use this image instead and remove the commands from the image:

**alpine-rng-tools-simple.yaml**

| apiVersion: apps/v1  kind: DaemonSet  metadata:  name: rngd  labels:  app.kubernetes.io/name: rngd  spec:  selector:  matchLabels:  app.kubernetes.io/name: rngd  template:  metadata:  labels:  app.kubernetes.io/name: rngd  spec:  affinity:  nodeAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  nodeSelectorTerms:  **-** matchExpressions:  **-** key: kubernetes.io/os  operator: In  values:  **-** linux  containers:  **-** name: rngd  image: us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-simple:1.0  securityContext:  capabilities:  add: ["SYS\_ADMIN"] |
| --- |

Finally, let's delete the workaround version of the DaemonSet and create a new one using our simple image:

| $ kubectl delete -n default daemonsets.apps rngd  daemonset.apps "rngd" deleted    $ kubectl apply -n default -f alpine-rng-tools-simple.yaml  daemonset.apps/rngd created    $ kubectl logs -n default -l app.kubernetes.io/name=rngd --tail=99  Initializing available sources  [hwrng ]: Initialization Failed  [rdrand]: Enabling RDSEED rng support  [rdrand]: Initialized  [jitter]: Initializing AES buffer  [jitter]: Enabling JITTER rng support  [jitter]: Initialized  unable to adjust write\_wakeup\_threshold: Read-only file system  ... |
| --- |

As we saw earlier the rngd daemon initializes and runs successfully.

# Advanced Minimal Image Approach

To build a more minimal image, without a shell or other utilities, we can perform a multi-stage image build process.

In the first stage we perform the same steps as before, except after installing the rng-tools package we gather the files from all the necessary packages and stage them in a temporary directory /tmp/alpine-rng-tools/.

The second stage, based on the empty scratch image, copies from the temporary staging directory into the final image. This will create a very small final image containing only the files from the necessary package dependencies.

**Dockerfile.multi-stage**

| FROM alpine:latest AS alpine-rng-tools-build  USER 0  RUN set -eu -o pipefail; \  apk update \  && apk upgrade \  && apk add rng-tools \  && apk info --contents alpine-baselayout musl libcrypto1.1 jitterentropy-library rng-tools \  | grep -Ev '^$|contains:$|/$' \  | while read a; do \  mkdir -p -m $(stat -c '%a' "${a%/\*}") "/tmp/alpine-rng-tools/${a%/\*}"; \  cp -afT "${a}" "/tmp/alpine-rng-tools/${a}"; \  done    FROM scratch AS alpine-rng-tools  COPY --from=alpine-rng-tools-build /tmp/alpine-rng-tools /  CMD ["rngd", "--foreground"] |
| --- |

We now build, tag, and push this image as before (the push had been done before so the layer already exists):

| $ docker build -t alpine-rng-tools-multi-stage:1.0 -f Dockerfile.multi-stage .  Sending build context to Docker daemon 3.584kB  Step 1/6 : FROM alpine:latest AS alpine-rng-tools-build  ---> 14119a10abf4  Step 2/6 : USER 0  ---> Using cache  ---> 5cefb65d7fe5  Step 3/6 : RUN set -eu -o pipefail; apk update && apk upgrade && apk add rng-tools && apk info --contents alpine-baselayout musl libcrypto1.1 jitterentropy-library rng-tools | grep -Ev '^$|contains:$|/$' | while read a; do mkdir -p -m $(stat -c '%a' "${a%/\*}") "/tmp/alpine-rng-tools/${a%/\*}"; cp -afT "${a}" "/tmp/alpine-rng-tools/${a}"; done  ---> Using cache  ---> 758494434f43  Step 4/6 : FROM scratch AS alpine-rng-tools  --->  Step 5/6 : COPY --from=alpine-rng-tools-build /tmp/alpine-rng-tools /  ---> Using cache  ---> 7ef62cd9483d  Step 6/6 : CMD ["rngd", "--foreground"]  ---> Using cache  ---> 3657efcbcf95  Successfully built 3657efcbcf95  Successfully tagged alpine-rng-tools-multi-stage:1.0    $ docker tag alpine-rng-tools-multi-stage:1.0 us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-multi-stage:1.0    $ docker push us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-multi-stage:1.0  The push refers to repository [us-west1-docker.pkg.dev/example-project/example-repo/alpine-rng-tools-multi-stage]  e7ef5f61f7e4: Layer already exists  1.0: digest: sha256:9ff0a908fad4d02b440a3340c94734716f0e2514b5fb1f6ed5ed6cccc043ce98 size: 528 |
| --- |

Again, the DaemonSet can be reconfigured (not shown) to use the alpine-rng-tools-multi-stage:1.0 image instead of the alpine-rng-tools-simple:1.0 image.

Building a custom minimal image reduces the attack surface of the image and decreases the size of the image by over 40%, from about 6MB down to about 3.5MB.

| $ (docker create --name="tmp\_$$" alpine-rng-tools-simple:1.0 >/dev/null && docker export tmp\_$$ && docker rm tmp\_$$ >/dev/null) | wc -c  6028288    $ (docker create --name="tmp\_$$" alpine-rng-tools-multi-stage:1.0 >/dev/null && docker export tmp\_$$ && docker rm tmp\_$$ >/dev/null) | wc -c  3534848 |
| --- |

# Conclusion

We showed three approaches to solving pre-5.6 Linux kernel entropy issues, from a simple DaemonSet using a public Alpine Linux base image, to generating a simple custom image, and finally to creating a minimal multi-stage image with reduced attack surface.

Hopefully you have found these techniques useful and can apply similar approaches to your specific problem domain.