## Selinux with Kubernetes Troubleshooting

You launch a container into Kubernetes and you get the message "CrashLoopBackOff". After a lot of troubleshooting you realize that it is Selinux blocking the container from starting. You have a few options to fix this issue.

1. Turn off Selinux, but then of course you lose all the security benefits of Selinux

2. Run a tool like audit2allow and simply allow the container to do whatever it wants to do. This is better than option 1, but will now allow all containers to run with those privileges which is likely not what we want.

3. Create a new policy that applies only to a specific container.

Luckily Red Hat has added a tool called Udica which makes the process of creating exceptions a little bit easier by allowing us to extend a few default polices. This post will step you through the process of troubleshooting and correcting these issues using the third strategy.

Our environment is Rancher's RKE2 running on Red Hat 8. The process should be similar on all distributions of Kubernetes, but might require some minor changes. In order to troubleshoot these issues we need to first install these packages on the individual nodes.

yum install -y setools-console policycoreutils-python-utils udica

### Selinux sample daemonset

For the purpose of demonstrating troubleshooting. daemonset which does a lot of things that no container should ever do to demonstrate the process. This daemonset runs Nginx and serves out /var/log on the underlying node through host path as well as writes it logs to this location as well. The daemonset also utilizes hostNetwork: true simply to prove a point. Finally the container runs as root. There really isn't a great reason to do this other than to create a few Selinux events that we need to troubleshoot. You should likely do this on a test cluster for obvious reasons.

You can view the daemonset and configmap at github <insert link here>

In order to run this, we recommend creating a seperate namespace. If Pod Security Policies are enabled you will need to assign the namespace an unrestricted policy due to the fact that we are violating a number of security rules.

Lets begin by setting up our environment! We are going to assume that Selinux is set to enforcing

kubectl create ns test-selinux

kubectl -n test-selinux create rolebinding unrestricted-psp --clusterrole=system-unrestricted-psp-role --group=system:serviceaccounts

git clone https://github.com/sealingtech/kubernetes\_selinux\_example

cd selinux-demo

kubectl apply -n test-selinux -f configmap.yml

kubectl apply -n test-selinux -f daemonset.yml

kubectl get pods -n test-selinux -o wide

When you view the pods, you will see they are failing. Lets logon to a node and figure out why. When you log in to the node and run the ausearch we can see that there are selinux messages in the logs blocking the container. Running the command we see that an error is being generated and AVC is denied but we might not be seeing all the possible events. It is possible that there are additional blocks that would take place, but that code is never being executed because the container is being killed before that code is ever executed.

ausearch -m avc -ts recent

Logon the to a single node and temporarily set one of the nodes to “permissive” mode which will set Selinux to no longer block but continue to log events. This will ensure we can see all events that would be generated while still allowing the code to be executed.

setenforce permissive

Delete the failing pod that is running on the node which you ran the setenforce command on. If you check the status of the pod on that node the pod should now be running.

kubectl delete pods -n test-selinux selinux-demo-nx24q

Lets open a web browser and go to:

http://<ip address of the host>:8080/

You should see a directory of /var/log of the underlying node. Click on the hawkey.log to generate an additional event.

On the individual node which the pod is running lets look at the events we generated. Lets look at two of the events.

# ausearch -m avc -ts recent

avc: denied { **open** } for pid=66790 comm="nginx" path="/srv/access.log" dev="dm-7" ino=20332 **scontext=system\_u:system\_r:container\_t:s0:c247,c262** **tcontext=system\_u:object\_r:var\_log\_t:s0** tclass=file **permissive=1**

node=edged-1.edged.dtramats.sealingtech.com type=AVC msg=audit(1669214633.942:4041): avc: denied { **append** } for pid=66790 comm="nginx" name="access.log" dev="dm-7" ino=20332 **scontext=system\_u:system\_r:container\_t:s0:c247,c262** **tcontext=system\_u:object\_r:var\_log\_t:s0 tclass=file** **permissive=1**

1. The subject (subj) is of the type container\_t. This is the default type that is used by Kubernetes unless otherwise specified. The security level (c247,c262) will be unique to each running container which isolates all the containers between one another running as the same type.
2. We see that if we were in enforcing mode we would deny the process nginx from open and appending to the target
3. The target (tcontext) is what is being acted on. The label for this is var\_log\_t.
4. Tclass specifies the type of object that the target is (file, directories, sockets, etc).
5. Permissive=1 means that this event was only logged and was allowed to go through.

So to put this another way a process running as the container\_t type is attempting to open and append to a file with the type var\_log\_t and would be denied if we were in enforcing mode.

Looking at the hawkey.log file we see that:

A process running as the container\_t type is attempting to open and read a file with the type rpm\_log\_t and would be denied if we were in enforcing mode.

On the node if we look at the labels in /var/log on the node we see that the label is in fact var\_log\_t.

ls -laZ /var/log

total 725536

drwxr-xr-x. 15 root root system\_u:object\_r:var\_log\_t:s0 4096 Nov 23 09:12 .

### The wrong way to fix this issue

So how can we go about fixing it? Lets first look at the naïve way which is simply allowing container\_t to write to var\_log\_t. If you were to utilize audit2allow to automatically generate rules we can see the rule to allow this.

ausearch -m avc -ts recent | audit2allow -a

#============= container\_t ==============

allow container\_t rpm\_log\_t:file { open read };

allow container\_t var\_log\_t:file { append open };

So audit2allow simply wants to allow this by allowing container\_t to append and open to var\_log\_t and open and read to rpm\_log\_t. The issue with this should be obvious. If we were to simply enable this policy all containers running can now perform these actions. That is likely NOT what we want because the container\_t type is the default that all containers run as container\_t unless otherwise specified.

Another method we could implement is to relabel the files to something that container\_t is able to do these actions to. Running sesearch will allow us to see all the rules that are defined by the source container\_t:

sesearch -A -s container\_t

…

allow container\_domain container\_file\_t:file { append create entrypoint execute execute\_no\_trans getattr ioctl link lock map mounton open read relabelfrom relabelto rename setattr unlink write };

…

With this we see that container\_t is able to write to files of container\_file\_t so if we wanted to allow hawkey.log to be read from all containers of the container\_t type we could simply relabel this file. This method is slightly more targeted as it now only allows container\_t to read this single file but still isn’t restricted enough. There is also a risk that something else could break which was expecting these files to be running as this context.

semanage fcontext -a -t container\_file\_t /var/log/hawkey.log

restorecon -v /var/log/hawkey.log

### The correct fix: Running containers as a different context.

The correct way to fix this is to extend the container\_t policy into a new policy which allows these actions to be performed and then add the few exceptions we need. Udica allows us to extend this easily. There is a default policy net\_container which allows containers to listen on a raw socket, we will extend this one.

Modify the daemonset to run the container as the testselinux\_t type. Add the following selinux to the securitycontext.

securityContext:

allowPrivilegeEscalation: true

runAsUser: 0

seLinuxOptions:

type: testselinux\_t.process

Lets delete the pod and open the browser and once again download the hawkey.log file. Lets see the new rules to allow this. Notice that these rules are now more targeted only effecting container\_t:

ausearch -m avc -ts recent | audit2allow -a

#============= testselinux\_t ==============

allow container\_t rpm\_log\_t:file { open read };

allow container\_t var\_log\_t:file { append open };

#============= testselinux\_t.process ==============

allow testselinux\_t.process http\_cache\_port\_t:tcp\_socket name\_bind;

allow testselinux\_t.process node\_t:tcp\_socket node\_bind;

allow testselinux\_t.process self:tcp\_socket listen;

After a few iterations of running the container in permissive, and looking through the logs and creating rules and testing I end up with the following rules to allow this container to run.

cat << EOF > /root/test-selinux.cil

(block testselinux\_t

(blockinherit container)

(allow process rpm\_log\_t ( file ( open read )))

(allow process var\_log\_t ( file ( append write open create )))

(allow process var\_log\_t ( dir ( write add\_name )))

(allow process http\_cache\_port\_t ( tcp\_socket ( name\_bind )))

(allow process node\_t ( tcp\_socket ( node\_bind )))

(allow process self ( tcp\_socket ( listen )))

)

EOF

semodule -i /root/test-selinux.cil /usr/share/udica/templates/base\_container.cil

setenforce enforcing

Apply this to all nodes on your cluster and then delete the pods. They should be now running. If we go to the node and run pstree -Z we should see that it is now running as testselinux\_t as we configured it.

├─containerd-shim(`system\_u:system\_r:container\_runtime\_t:s0')

│ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ ├─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

│ │ └─nginx(`system\_u:system\_r:testselinux\_t.process:s0:c629,c931')

With this we have created a properly confined container that runs inside of testselinux\_t and is able to run as we expected. This means that only containers running as this specific type is able to perform these specific tasks which follows least privilege principles.