Anubolu Bharath
8 min read · Just now

Improperly configured pods and containers can be exploited to attack the Kubernetes cluster and other applications. MITRE ATT&CK® framework, which is a de-facto industry standard for describing threats lists threat vectors such as Execution, Persistence, Privilege escalation and Credential access that can exploit inadequately secured pods and containers. Fortunately, Kubernetes provides several native mechanisms to improve security of your pods and containers, one of them being Security Context.

Security Context

A security context defines the operating system security settings, such as controlling whether a process can gain more privileges than its parent process, restricting write access to container's root filesystem, file-like object access permissions based on uid and gid, Linux capabilities, SELinux role, etc., applied to a pod or individual container. The constraints imposed by a security context are designed to achieve the following goals:

1. Ensure a clear isolation between container and the underlying host it runs on
2. Limit the ability of the container to negatively impact the infrastructure or other containers

Security context can be set at pod `PodSecurityContext` or container `SecurityContext` levels. Equivalent fields at the container level take precedence over the fields specified in the `PodSecurityContext`. One thing to note is that `PodOS` field in the `PodSpec` can restrict some security context fields at both pod and container level.

Putting it into practice

To try the examples, you need to have a Kubernetes cluster and the `kubectl` command line tool that is configured to communicate with it. If you do not have a

cluster at hand, use an online sandbox such as Killercoda or create your own Kubernetes cluster using a tool like minicube.

Set process and volume mount permissions

First, let us see what security applies to a pod with no security context set.

Regular privileged pod

Save the following manifest to a file named `regular-pod.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: regular-pod-demo
spec:
  volumes:
  - name: demo-volume
    emptyDir: {}
  containers:
  - name: regular-demo
    image: redhat/ubi8
    command: [ "tail", "-f", "/dev/null" ]
    volumeMounts:
    - name: demo-volume
      mountPath: /data/demo
```

Create the pod and create an interactive session into it.

```
# Create the pod
kubectl create -f regular-pod.yaml
```

```
# Verify that the pod is running
kubectl get pod/regular-pod-demo
NAME                READY   STATUS    RESTARTS   AGE
regular-pod-demo    1/1     Running   0          15s

# Exec into the pod
kubectl exec -it regular-pod-demo -- sh
sh-4.4#
```

Check the `uid` and `gid` of the `tail` process that is running in the container. You can verify that this is indeed the command that we have specified earlier in the `command` field by running `cat /proc/1/cmdline`

```
stat -c "%u %g" /proc/1/
0 0
```

As we can see, the command runs as the root user with `uid=0` and `gid=0`.

A container running without a security context has a number of capabilities enabled by default. These can be checked using `capsh` command, which is provided by `libcap2-bin` package.

```
capsh --print
Current: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,c
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_set
Ambient set =
Current IAB: !cap_dac_read_search,!cap_linux_immutable,!cap_net_broadcast,!cap_
Securebits: 00/0x0/1'b0 (no-new-privs=0)
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=
Guessed mode: UNCERTAIN (0)
```

But what about the mounted volume?

```
ls -l /data
total 4
drwxrwxrwx 2 root root 4096 Jun  8 21:13 demo
```

As expected, the volume is owned by the root user and group. Any files created on the volume will have matching permissions.

Non-privileged pod

What is going to happen if all privileges are removed from the container? Let us create a non privileged pod and remove all Linux capabilities from it. Below is the contents of `non-privileged-pod.yaml` manifest.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-privileged-pod-demo
spec:
  volumes:
  - name: demo-volume
    emptyDir: {}
  containers:
  - name: non-privileged-demo
    image: redhat/ubi8
    command: [ "tail", "-f", "/dev/null" ]
    volumeMounts:
    - name: demo-volume
      mountPath: /data/demo
    securityContext:
      privileged: false
      allowPrivilegeEscalation: false
      capabilities:
        drop:
          - ALL
```

Create the pod and exec into it.

```
kubectl create -f non-privileged-pod.yaml
kubectl exec -it non-privileged-pod-demo -- sh
```

Let us inspect the capabilities of the container this time.

```
capsh --print
Current: =
Bounding set =
Ambient set =
Current IAB: !cap_chown,!cap_dac_override,!cap_dac_read_search,!cap_fowner,!cap
Securebits: 00/0x0/1'b0 (no-new-privs=1)
```

```
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root) euid=0(root)
gid=0(root)
groups=
Guessed mode: UNCERTAIN (0)
```

```
# Alternatively, capabilities can be inspected by running
cat proc/1/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000000000000000
CapAmb: 0000000000000000
```

As indicated by `Current: =` and `Bounding set =` lines in the `capsh` output, all capabilities have been removed from the container. In practice that means that operations that require privileged access are not permitted even for the root user.

```
su -
su: cannot set groups: Operation not permitted
```

```
yum install -yq nmap-ncat
Error unpacking rpm package libibverbs-44.0-2.el8.1.x86_64
Error unpacking rpm package libpcap-14:1.9.1-5.el8.x86_64
Error unpacking rpm package nmap-ncat-2:7.70-8.el8.x86_64
Failed:
  libibverbs-44.0-2.el8.1.x86_64                          libpcap-
14:1.9.1-5.el8.x86_64                          nmap-ncat-2:7.70-
8.el8.x86_64
Error: Transaction failed
```

The root user, however, can still do a lot of accidental or intentional damage to the system even in unprivileged container. For example, the root user can delete the contents of the root file system.

```
rm -rf /*
```

```
ls
sh: /usr/bin/ls: /usr/bin/coreutils: bad interpreter: No such file
or directory
```

Even for a non-privileged container we **must not run processes as a privileged user.**

Non-privileged pod with non-root user

This example uses a different image as it needs the `iputils` package.

It is possible to enforce the use of non-root users by containers setting `runAsNonRoot` parameter to `true`.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-root-user-pod-demo
spec:
  securityContext:
    runAsNonRoot: true
  volumes:
  - name: demo-volume
    emptyDir: {}
  containers:
  - name: non-root-user-demo
    image: demisto/iputils:1.0.0.62867
    command: [ "tail", "-f", "/dev/null" ]
    volumeMounts:
    - name: demo-volume
      mountPath: /data/demo
    securityContext:
      privileged: false
      allowPrivilegeEscalation: false
      capabilities:
        drop:
          - ALL
```

The container image used in this example defaults to the root user, so it should fail to start after the constraint was applied. Let us verify that.

```
kubectl create -f runasnonroot.yaml
kubectl get pods/runasnonroot-pod-demo -ojson | jq .status.containerStatuses
[
  {
    "image": "demisto/iputils:1.0.0.62867",
    "imageID": "",
    "lastState": {},
    "name": "non-root-user-demo",
    "ready": false,
    "restartCount": 0,
    "started": false,
    "state": {
      "waiting": {
        "message": "container has runAsNonRoot and image will run as root (pod:
        "reason": "CreateContainerConfigError"
      }
    }
  }
]
```

As expected, the container failed to start. To fix the container, either specify a non-root user during the container build time, or use `runAsUser` parameter as discussed below.

To make the processes running inside a pod to be owned by a specific user and group requires setting `runAsUser` and `runAsGroup` fields. This can be done at either the pod or the container level. As mentioned previously, container-level security context takes precedence over pod one. Save the manifest as `non-root-user-pod.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-root-user-pod-demo
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: demo-volume
    emptyDir: {}
  containers:
```

```
    - name: non-root-user-demo
      image: demisto/iputils:1.0.0.62867
      command: [ "tail", "-f", "/dev/null" ]
      volumeMounts:
      - name: demo-volume
        mountPath: /data/demo
      securityContext:
        privileged: false
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
```

Create container and start an interactive session.

```
kubectl create -f non-root-user-pod.yaml
kubectl exec -it non-root-user-pod-demo -- sh
```

We can now verify that the user has an unprivileged `uid` and `gid` and has a reduced level of access to the container's root file system.

```
# Permissions of the default process that is running inside the container
stat -c "%u %g" /proc/1/
1000 3000
```

```
# Displaying permissions of the current user
id
uid=1000 gid=3000 groups=3000,2000

# A non-root user has reduced access to the container's root file
system
touch /aa
touch: cannot touch '/aa': Permission denied
```

Note that in addition to the primary group `3000` the user is also a member of an auxiliary group `2000` that we have specified as `fsGroup` in the manifest. The `fsGroup` controls file permissions of the mounted volume.

```
# The volume is owned by the auxiliary group (gid=2000)
ls -l /data
total 4
drwxrwsrwx 2 root 2000 4096 Jun  9 08:35 demo
```

```
# New files on the volume are owned by the current user and the
auxiliary group id
touch /data/demo/aa
sh-4.4$ ls -l /data/demo/aa
-rw-r--r-- 1 1000 2000 0 Jun  9 08:40 /data/demo/aa

# Files created elsewhere, still have gid equal to the current
user's gid
touch /tmp/aa
ls -l /tmp/aa
-rw-r--r-- 1 1000 3000 0 Jun  9 08:45 /tmp/aa
```

The container security has improved by configuring a security context that removes privileges and ensures that the processes inside the container run as unprivileged user. What if we need some privileged access to run a console command?

```
ping google.com
ping: socktype: SOCK_RAW
ping: socket: Operation not permitted
ping: => missing cap_net_raw+p capability or setuid?
```

Run ping in a non-privileged container

It is possible to grant specific privileges to an otherwise unprivileged pod. In the previous example we could not run `ping` from an unprivileged pod. To find which privileges are required by an executable, use `getcap` command. Ping requires `CAP_NET_RAW` Linux capability to run and `allowPrivilegeEscalation` set to `true`. When specifying capabilities in the container manifest, remember to omit the `CAP_` portion of the constant. Save the following manifest as `capability-pod.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: capability-pod-demo
```

```
  spec:
    securityContext:
      runAsNonRoot: true
      runAsUser: 1000
      runAsGroup: 3000
      fsGroup: 2000
    volumes:
    - name: demo-volume
      emptyDir: {}
    containers:
    - name: capability-demo
      image: demisto/iputils:1.0.0.62867
      command: [ "tail", "-f", "/dev/null" ]
      volumeMounts:
      - name: demo-volume
        mountPath: /data/demo
      securityContext:
        privileged: false
        allowPrivilegeEscalation: true
        capabilities:
          drop:
            - ALL
          add:
           - NET_RAW
```

Verify that we can run the `ping` command.

```
# Create the pod
kubectl create -f capability-pod.yaml
```

```
# Execute ping command
kubectl exec -it capability-pod-demo -- ping google.com
PING google.com (172.217.18.110) 56(84) bytes of data.
64 bytes from fra16s42-in-f14.1e100.net (172.217.18.110): icmp_seq=1
ttl=110 time=1.45 ms
64 bytes from fra16s42-in-f14.1e100.net (172.217.18.110): icmp_seq=2
ttl=110 time=1.52 ms
```

Enforce read-only root file system

Read-only root file system helps to enforce immutable infrastructure strategy. Most containers are stateless and should not need to write to the local file system. Those containers that do, should write on the mounted volumes that persist the container restart or termination.

Ensuring container root file system is immutable and employing a verified boot mechanism offers protection against potential attackers gaining control over the host machine through permanent local changes. It also safeguards the host system from malicious binaries attempting to unauthorised writes to the host file system.

At the container level, the securityContext includes a parameter called `readOnlyRootFilesystem`, which, when set to `true`, mounts the root file system in read-only mode. To demonstrate this, let's modify the initial example to set the root file system as read-only and subsequently test its behaviour.

```
apiVersion: v1
kind: Pod
metadata:
  name: read-only-root-fs-pod-demo
spec:
  containers:
  - name: read-only-root-fs-demo
    image: redhat/ubi8
    command: [ "tail", "-f", "/dev/null" ]
    securityContext:
      readOnlyRootFilesystem: true
```

Save the manifest as `read-only-root-fs.yaml` and run the following commands.

```
# Create the pod
kubectl create -f read-only-root-fs.yaml
```

```
# Attempt to create a file
kubectl exec -it read-only-root-fs-pod-demo -- touch aa
touch: cannot touch 'aa': Read-only file system
command terminated with exit code 1
```

Conclusions

We have learned the importance of securing containers and pods, as they can be exploited to attack the Kubernetes cluster and other applications running on it. By default pods have a lot of capabilities enabled and run processes as the root user. Security context provides a mechanism to create unprivileged pods, make root file

system read-only and run processes in containers as non-root users. Linux capabilities can be selectively added to containers to allow execution of processes that require them.

Edit profile

Written by Anubolu Bharath

0 Followers · 1 Following

No responses yet

What are your thoughts?

Respond

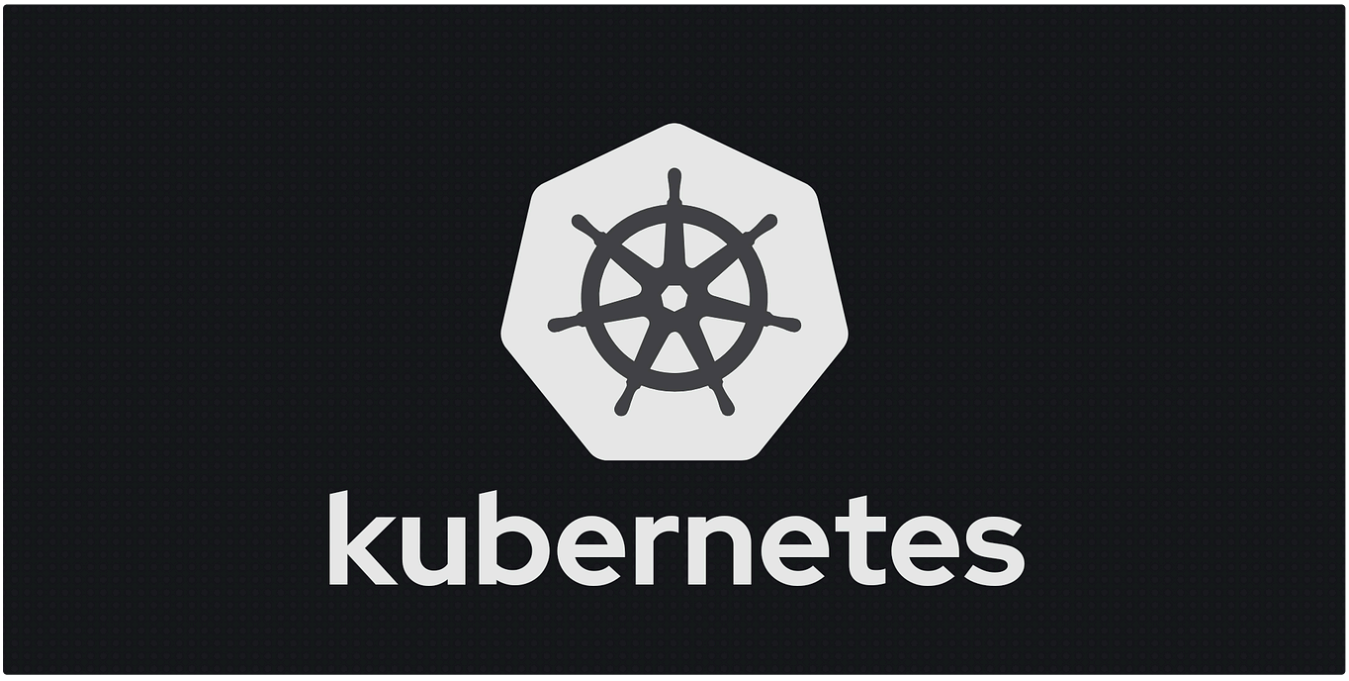
Recommended from Medium

Jerome Decinco

Tuned in Linux: Optimizing System Performance with Profiles

Managing system performance in Linux can be tricky, especially when juggling between various workloads like databases, virtualization, or...

6d ago 51



In Stackademic by Crafting-Code

I Stopped Using Kubernetes. Our DevOps Team Is Happier Than Ever

Why Letting Go of Kubernetes Worked for Us

Nov 19 3.6K 110

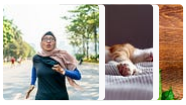
Lists

Staff picks

778 stories · 1484 saves

Stories to Help You Level-Up at Work

19 stories · 887 saves



Self-Improvement 101

20 stories · 3111 saves

Productivity 101

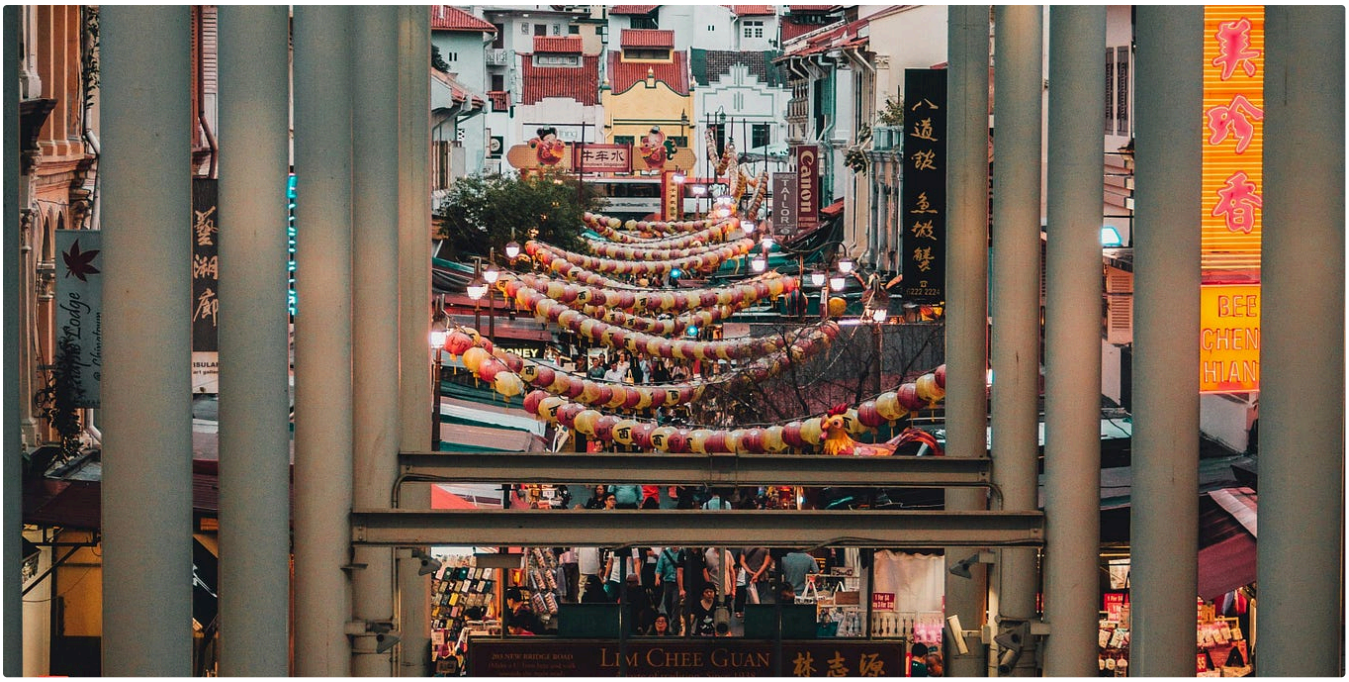
20 stories · 2615 saves

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 6.2K 89



Mark Shrime, MD, PhD

The dumbest decision I ever made (and the Nobel Prize that explains it)

Decision science, Family Guy, and why the “safe” choice is often the riskiest.

Nov 21 7.7K 182

Jessica Stillman

Jeff Bezos Says the 1-Hour Rule Makes Him Smarter. New Neuroscience Says He’s Right

Jeff Bezos’s morning routine has long included the one-hour rule. New neuroscience says yours probably should too.

See more recommendations