10 Kubernetes Security Context settings you should understand

Securely running workloads in Kubernetes can be difficult. Many different settings impact Kubernetes API security, requiring significant knowledge to implement correctly. One of the most powerful tools Kubernetes provides in this area are the `securityContext` settings that every Pod and Container manifest can leverage. In this cheatsheet, we will take a look at the various `securityContext` settings, explore what they mean and how you should use them.

1. runAsNonRoot
2. runAsUser / runAsGroup
3. seLinuxOptions
4. seccompProfile
5. privileged / allowPrivilegeEscalation
6. capabilities
7. readonlyRootFilesystem
8. procMount
9. fsGroup / fsGroupChangePolicy
10. sysctls

Pod vs Container settings

Kubernetes `securityContext` settings are defined in both the **PodSpec** and **ContainerSpec APIs**, and the scoping is indicated in this document by the [P] and/or [C] annotations next to each one. Note that if a setting is available and configured in both scopes the container setting will take precedence.

Now, in no particular order, let's take a look at the `securityContext` settings:

1. runAsNonRoot [P/C]

Even though a container uses **namespaces** and **cgroups** to limit its process(es), all it takes is one misconfiguration in its deployment settings to grant those processes access to resources on the host. If that process runs as root, it has the same access as the host root account to those resources. Additionally, if other pod or container settings are used to reduce constraints (i.e. procMount or capabilities), having a root UID compounds the risks of any exploitation of them. Unless you have a very good reason, you should never run a container as root.

So, what do you do if you have an image to deploy that is using root?

OPTION 1: USE THE USER PROVIDED IN THE BASE IMAGE

Often, base images will already have a user created and available but leave it up to the development or deployment teams to leverage it. For example, the official **Node.js image** comes with a user named node at UID 1000 that you can run as, but they do not explicitly set the current user to it in their Dockerfile. We will either need to configure it at runtime with a **runAsUser** setting or change the current user in the image using a derivative Dockerfile. The former assumes that UID 1000 can read the files in the `appvolume` mount, and it also is not a very common use case to have the application in a volume. Let's instead look at an example using a derivative Dockerfile to build our own image.

Without diving too deep into image building, let's assume we have a pre-built npm application. Here is a minimal Dockerfile to build an image based on `node:slim` and run as the provided node user.

```
FROM node:slim
COPY --chown=node . /home/node/app/   # <--- Copy app into the home directory with right ownership
USER 1000                             # <--- Switch active user to "node" (by UID)
WORKDIR /home/node/app                # <--- Switch current directory to app
ENTRYPOINT ["npm", "start"]           # <--- This will now exec as the "node" user instead of root
```

The key line starts with USER which makes node the default user inside any container started from this image. We use the UID instead of the name of the user because Kubernetes cannot map an image's default user name to its UID before starting the container and will return an error about this when deploying with `runAsNotRoot:` true specified.

OPTION 2: BASE IMAGE PROVIDES NO USER

So what would we do if the node base image did not already provide us with a user to use? For many processes, we simply create one in a derivative Dockerfile and use it. Let's extend the previous example to do that:

```
FROM node:slim
RUN useradd somebody -u 10001 --create-home --user-group  # <--- Create a user
COPY --chown=somebody . /home/somebody/app/
USER 10001
WORKDIR /home/somebody/app
ENTRYPOINT ["npm", "start"]
```

As you can see, the only addition is the `RUN` line that creates a user–the syntax of this may vary depending on the base image distro–and I've changed the user and path references to match it afterward.

NOTE: This works fine for node.js and npm but it is possible that other tools may require different elements of the filesystem to have ownership changes. Consult your tool's documentation if you encounter any issues.

2. runAsUser / runAsGroup [P/C]

Container images may have a specific user and/or group configured for the process to run as. This can be overridden with the `runAsUser` and `runAsGroup` configuration settings. Often these are set up in conjunction with volume mounts containing files that have the same ownership IDs.

```
...
spec:
  containers:
  - name: web
    image: mycorp/webapp:1.2.3
  securityContext:
    runAsNonRoot: true
    runAsUser: 10001
...
```

There is a danger in using these settings as you are making runtime decisions for the container that may not be compatible with the original image. For example, the `jenkins/jenkins` CI official server image runs as group:user named `jenkins:jenkins` and its application files are all owned by that user. If we configure a different user, it will fail to start up because that user doesn't exist in the image `/etc/passwd` file. Even if it somehow did, it's very likely to have problems reading and writing to the files owned by `jenkins:jenkins`. A simple `docker run` command can check this for you:

```
$ docker run --rm -it -u eric:eric jenkins/jenkins
docker: Error response from daemon: unable to find user eric: no matching entries in passwd file.
```

As we mentioned above, it is a very good idea to ensure container processes do not run as the root user but don't rely on the `runAsUser` or `runAsGroup` settings to guarantee this. Someone could remove these settings in the future. Be sure to also set **runAsNonRoot** to true.

3. seLinuxOptions [P/C]

SELinux is a policy driven system to control access to applications, processes and files on a Linux system. It implements the **Linux Security Modules** framework in the Linux kernel. SELinux is based on the concept of labels. It applies these labels to all the elements in the system which group elements together. These labels are known as the security context–not to be confused with the Kubernetes `securityContext`–and consist of a `user`, `role`, `type`, and an optional field `level` in the format `user:role:type:level.`

SELinux then uses policies to define which processes of a particular context can access other labelled objects in the system. SELinux can be strictly enforced, in which case access will be denied, or it can be configured in permissive mode where it will log access. In containers, SELinux typically labels the container process and the container image in such a way as to restrict the process to only access files within the image.

Default SELinux labels will be applied by the container runtime when instantiating a container. The seLinuxOptions setting in `securityContext` allows custom SELinux labels to be applied. Be aware that changing the SELinux labeling for a container could potentially allow the containerized process to escape the container image and access the host filesystem.

Note that this functionality will only apply if the host operating system supports SELinux.

4. seccompProfile [P/C]

Seccomp stands for [secure computing mode](#) and is a feature of the Linux kernel which can restrict the calls a particular process can make from user space into the kernel. A seccomp profile is a JSON definition typically consisting of a set of syscalls and the default action taken if one of these syscalls occurs.

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "accept",
            "action": "SCMP_ACT_ALLOW",
            "args": []
        },
        {
            "name": "accept4",
            "action": "SCMP_ACT_ALLOW",
            "args": []
        },
        ...
    ]
}
```

Example seccomp profile copied from https://training.play-with-docker.com/security-seccomp/

Kubernetes provides a mechanism for using custom profiles through the `seccompProfile` setting in `securityContext`.

```
seccompProfile:
      type: Localhost
      localhostProfile: profiles/myprofile.json
```

There are three possible values for the type field:

- `Localhost` with which a `localhostProfile` setting provides a path inside the container to a seccomp profile
- `Unconfined` in which no profile is applied.
- `RuntimeDefault` in which the container runtime default is used–this is the default if the type is left unspecified

You can apply these settings either in a PodSecurityContext or securityContext. If both are set, the settings at the container level in securityContext are used. Note that the securityContext configuration API was released in Kubernetes v1.19 – if you are deploying to earlier versions there is a different syntax; consult the **Kubernetes documentation site** for details and examples.

As with most security-related settings, the principle of least privilege applies here. Only give your container access to the privileges it needs and no more. Begin by creating a profile that simply logs which syscalls are taking place, then test your application to build a set of allowed syscalls. You can find more information on this process in the Kubernetes **tutorials**.

5. Avoid Privileged Containers / Escalations [C]

Granting a container privileged status is dangerous and is usually used as a simpler way to achieve specific permissions that can otherwise be controlled through granting capabilities access. The container runtime controls the exact implementation of the privileged flag, but it will effectively grant the container all privileges and lift limitations enforced by the device cgroup controller. It may also modify the Linux Security Module configuration, and allow for processes inside the container to escape the container.

Containers provide process isolation in the host, so even with the container running as root, there are capabilities that the container runtime does not grant to the container. With the privileged flag set, the container runtime grants all of the system root's capabilities, making it extremely dangerous from a security perspective since it allows full access to the underlying host system.

Avoid using the privileged flag, and if your container does need additional capabilities, add only the ones you need through the capabilities settings. Unless your container needs to control system level settings in the host kernel–like access to specific hardware or reconfiguring networks–and needs access to the host filesystem, then it does not need the privileged flag.

For a deeper dive into Privileged Containers, check out Matt's blog article: **Privileged Docker containers**—do you really need them?

6. Linux kernel capabilities [C]

Capabilities are **kernel level permissions** that allow for more granular controls over kernel call permissions than simply running everything as root. Capabilities include things like the ability to change file permissions, control the network subsystem, and perform system-wide administration functions. In `securityContext`, Kubernetes provides configuration to drop or add capabilities. Individual capabilities or a comma-separated list may be provided as a string array. Alternatively, you can use the `-all` shorthand to add or drop all capabilities. This configuration is passed down to the container runtime, configuring the capability set when it creates the container. If no capabilities section is present in `securityContext`, then the container is provided with the default set of capabilities that the container runtime provides.

```
securityContext:
      capabilities:
        drop:
          - all
        add: ["MKNOD"]
```

The recommended practice is to drop all capabilities, then only add back the ones your application actually needs. In many cases applications don't actually require any capabilities in normal operation, test this by dropping them all and debug any failures by monitoring the audit logs to see which capabilities have been blocked.

Note that when listing capabilities to drop or add in `securityContext`, you remove the `CAP_` prefix which the kernel uses in naming capabilities. For debugging purposes, the `capsh` tool will give you a human-readable output of exactly which capabilities are enabled in your container and is available for most distributions. Don't leave this available in production containers, as this makes it very easy for an attacker to work out which capabilities are enabled! If you really love to read bitmaps, you can also check the enabled capabilities in the `/proc/1/status` file.

Learn how to improve Kubernetes security by dropping **default capabilities for a container.**

7. Run with a read-only filesystem [C]

If your container gets compromised, and it has a read-write filesystem, an attacker is free to change its configuration, install software, and potentially launch other exploits. Having a read-only file system helps prevent these kinds of escalations by limiting the actions that an attacker can perform. In general, containers shouldn't require writing to the container filesystem. If your application has stateful data then you should be using an external persistence method such as a database, volume, or some other service. Also, ensure that all logs are written to stdout and/or a log forwarder where they can be collated centrally.

8. procMount [C]

By default, container runtimes mask certain parts of the `/proc` filesystem from inside a container in order to prevent potential security issues. However, there are times when access to those parts of `/proc` is required; particularly when using nested containers as is often used as part of an in-cluster build process. There are only two valid options for this entry: `Default`, which maintains the standard container runtime behavior, or `Unmasked`, which removes all masking for the `/proc` filesystem.

Obviously, you should only use this entry if you really know what you are doing. If you are using it for image building purposes, check the latest version of your build tool as many no longer need this. Upgrade and return to the default `procMount` that is true for the tool you are using.

Finally, if you do find that you are needing to use this, only do so for a nested container; never expose the `/proc` filesystem of your host system to a container.

9. fsGroup / fsGroupChangePolicy [P]

The `fsGroup` setting defines a group which Kubernetes will change the permissions of all files in volumes to when volumes are mounted by a pod. The behavior here is also controlled by the `fsGroupChangePolicy`, which can be set to `onRootMismatch` or Always. If set to `onRootMismatch` the permissions will only be changed if they don't already match the permissions of the container root.

Be cautious with the use of `fsGroup`. The changing of group ownership of an entire volume can cause pod startup delays for slow and/or large filesystems. It can also be detrimental to other processes that share the same volume if their processes do not have access permissions to the new GID. For this reason, some providers for shared file systems such as NFS do not implement this functionality. These settings also do not affect ephemeral volumes.

10. sysctls [P]

Sysctls are a function of the Linux kernel which allows administrators to modify kernel configuration. In a full Linux operating system, these are defined through the use of `/etc/sysctl.conf`, and can also be modified using the `sysctl` utility.

The `sysctls` setting in `securityContext` allows specific sysctls to be modified in the container. There are only a small subset of the operating system sysctls which can be modified on a per container basis that are namespaced in the kernel. Out of this subset, some are considered safe. A much bigger set are considered unsafe, depending on the potential for impacting other pods. The unsafe sysctls are generally disabled in clusters and need to be specifically enabled by the cluster administrator.

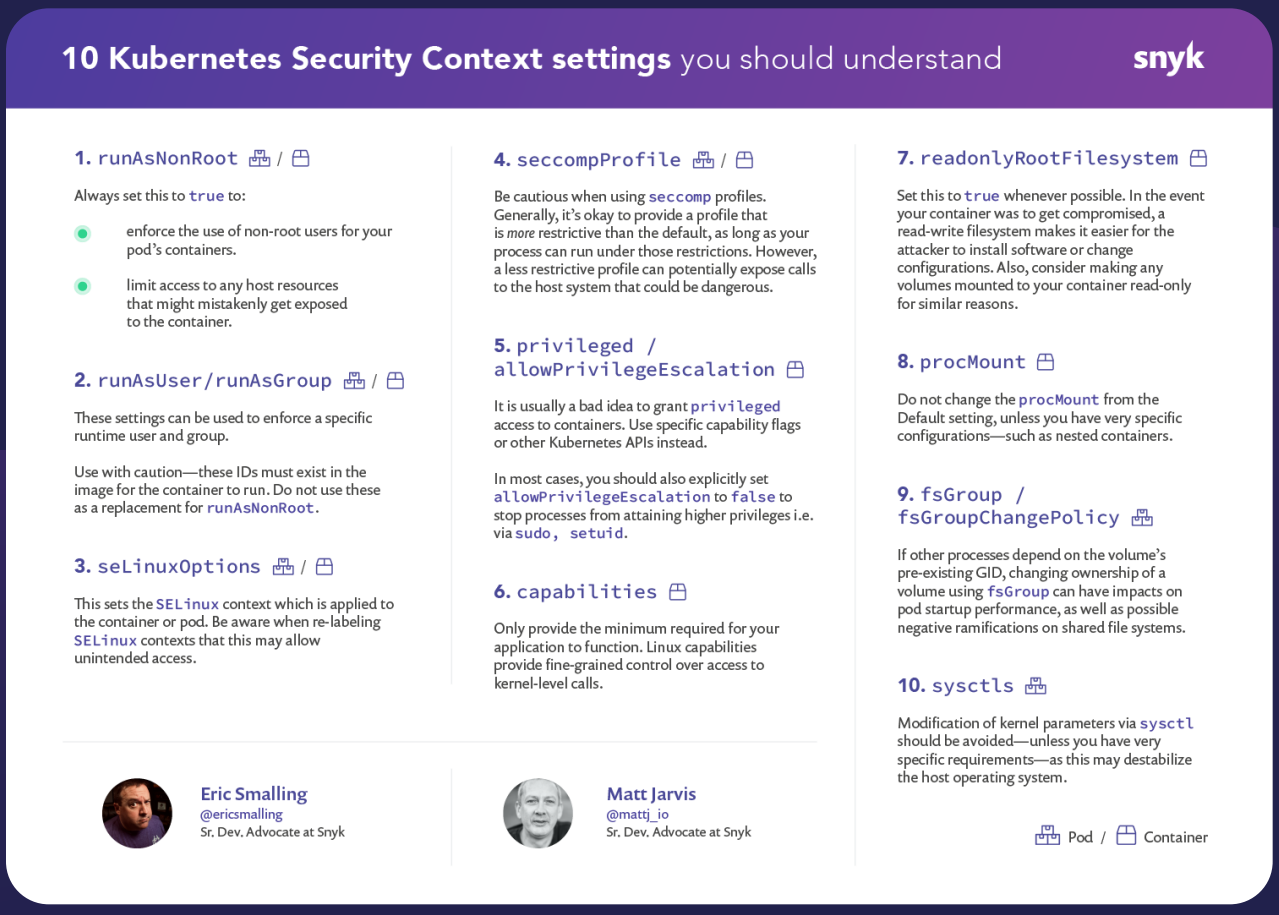
Given the potential for destabilizing the underlying operating system, modification of kernel parameters via `sysctls` should be avoided unless you have very specific requirements. You should also review such changes with your cluster operator.

A note about securityContext at runtime

In many cases, the security settings described here are combined with policy-based admission control to ensure that required settings are actually configured before launching containers into the cluster. By combining `securityContext` settings with a `PodSecurityPolicy`, you can ensure that only containers which follow the policy are launched by enforcing specific `securityContext` settings. securityContext settings can also be appended to container configuration at launch time through Dynamic Admission Control, and the use of mutating webhooks.

Conclusion

There are a lot of things to keep in mind when hardening your application deployments with `securityContext` settings. When used properly, they are a highly effective tool and we hope this list will help your teams choose the right options for your workloads and environments. Snyk can help you with these choices by scanning your Kubernetes yaml files for common misconfigurations. Sign up for your free account with the button below.



snyk