Jonathan

Follow 29 Followers About

Certified Kubernetes Security Specialist (CKS) Preparation Part 8 — Runtime Security & System Hardening

Jonathan Mar 24 · 8 min read

If you have not yet checked the previous parts of this series, please go ahead and check Part1, Part2, Part3, Part4, Part5, Part6 and Part7.

In this article, I would focus on the preparation around **runtime security** and **system hardening** in CKS certification exam.

strace

The full name of strace is system call trace, which means it is a process that would act like a sidecar on system call interface and note down every syscall. This Medium article gives a pretty straightforward explanation on how most people use strace in practice and this one would give a thorough walk-through if anyone is interested in the details. At the end of the day, there are so many parameters you could use with strace and all really depends on the situation. For demonstration, we would use the commonly used ones when debugging Kubernetes operations.

One of the examples is to see whether we could read secrets stored in etcd. In order to achieve that, we would need to know the process ID etcd is running on.

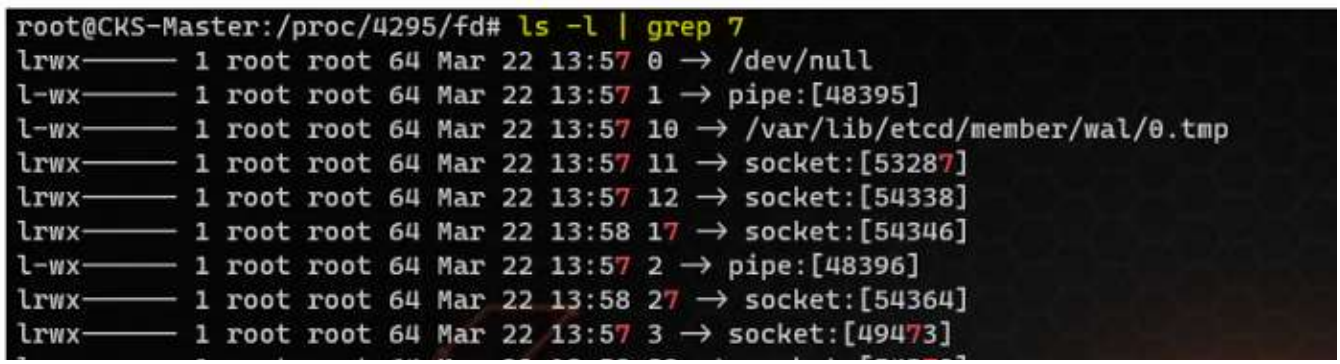
```
#SSH into K8s master nodes
- ps aux | grep etcd
```

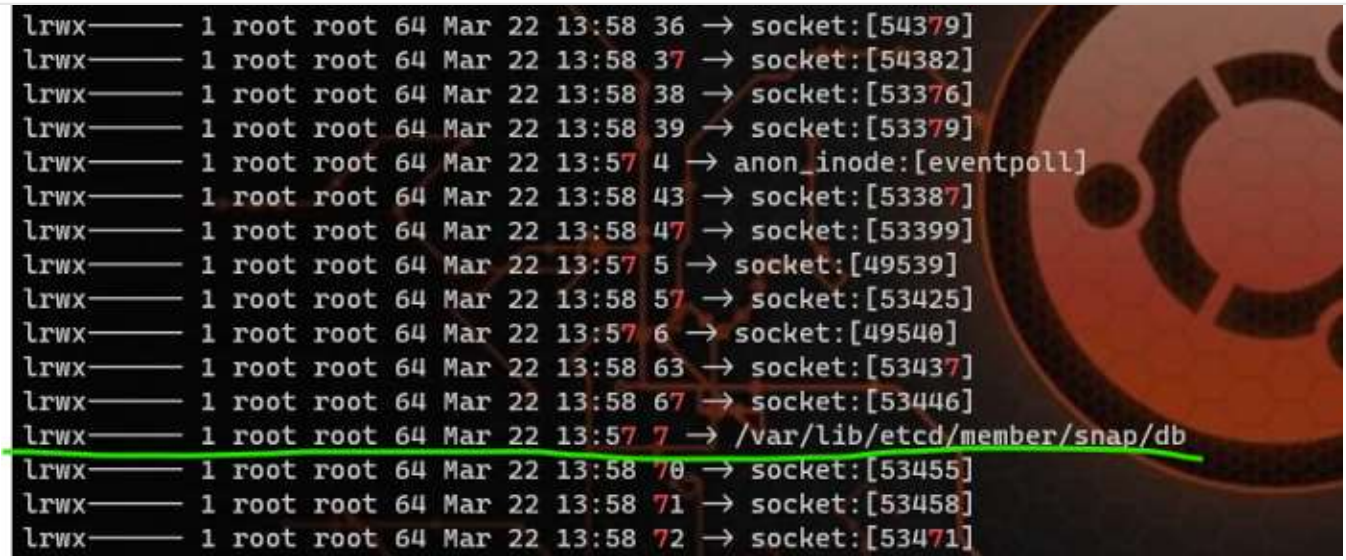
From the result above, the first process ID would be our target as the second process is kube-apiserver and the third is the “grep” we just execute.

```
sudo strace -p 4295
```

We should be seeing a lot of operations being listed out. From here, we could head over to the process directory and see what are contained within it.

```
- sudo su
- cd /proc/4295/fd
- ls -l | grep 7
```



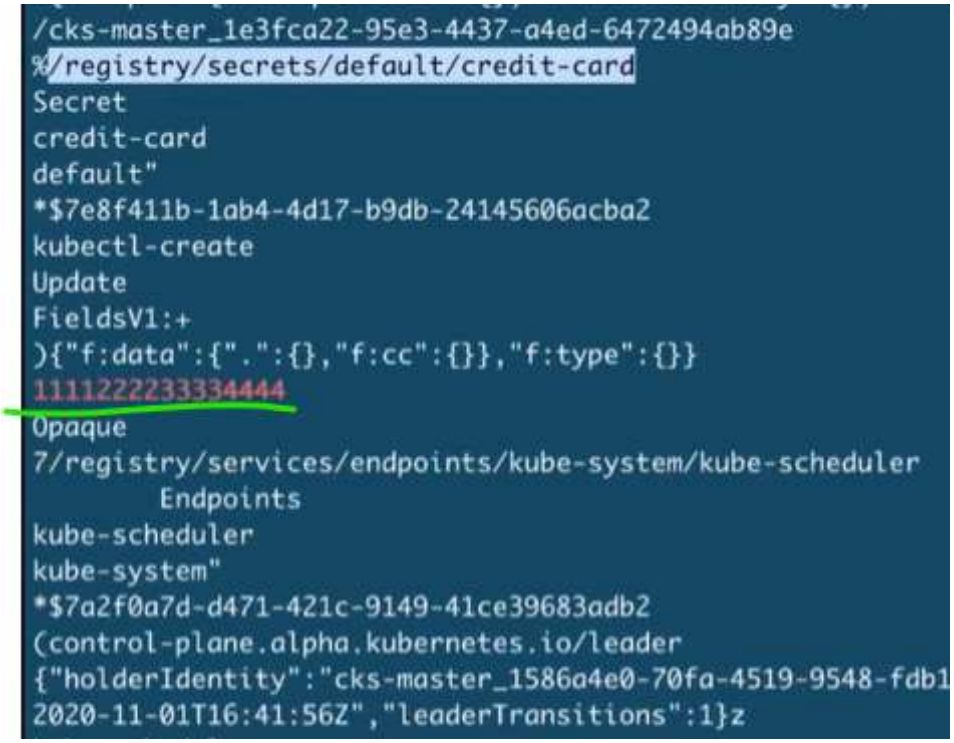


At this point, it seems like directory “7” contains the information that K8s needs. We could create a simple secret and try to locate the value within it.

```
- kubectl create secret generic credit-card --from-literal
ssecret=1111222233334444

#Make sure you are still in directory /proc/4295/fd.
#"-A10" and "-B10" mean show 10 lines before and after the searching
string.

- cat 7 | strings | grep 1111222233334444 -A10 -B10
```



Quoted from Udemy — Kubenetes CKS 2021 Complete Course

As my testing environment suddenly breaks, using the CKS course screen captures to show the expected result when getting into etcd's data.

Falco Installation and Use Scenarios

Falco is a CNCF project which is invented to trace all Kubernetes administrators' actions. In fact, it could be both a great auditing tool and also a rule engine to identify any violation.

Falco could be installed to K8s through standalone mode or daemonset mode (having a Pod running in every node). In this example, we would install this service through standalone mode. Head here for more details.

After installation, check the service status after starting.

```
root@CKS-Worker:/home/jonw# systemctl start falco
root@CKS-Worker:/home/jonw# systemctl status falco
● falco.service - LSB: Falco syscall activity monitoring agent
   Loaded: loaded (/etc/init.d/falco; generated)
   Active: active (running) since Sun 2021-02-07 16:51:57 UTC; 1s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 30778 ExecStart=/etc/init.d/falco start (code=exited, status=0/SUCCESS)
    Tasks: 10 (limit: 4915)
   CGroup: /system.slice/falco.service
           └─30809 /usr/bin/falco --daemon --pidfile=/var/run/falco.pid

Feb 07 16:51:57 CKS-Worker falco[30798]: Loading rules from file /etc/falco/falco_rules.yaml:
Feb 07 16:51:57 CKS-Worker falco[30798]: Loading rules from file /etc/falco/falco_rules.local.yaml:
Feb 07 16:51:57 CKS-Worker falco[30778]: Sun Feb  7 16:51:57 2021: Loading rules from file /etc/falco/falco_rules.local.yaml:
Feb 07 16:51:57 CKS-Worker falco[30798]: Loading rules from file /etc/falco/k8s_audit_rules.yaml:
Feb 07 16:51:57 CKS-Worker falco[30778]: Sun Feb  7 16:51:57 2021: Loading rules from file /etc/falco/k8s_audit_rules.yaml:
Feb 07 16:51:57 CKS-Worker systemd[1]: Started LSB: Falco syscall activity monitoring agent.
Feb 07 16:51:57 CKS-Worker falco[30809]: Starting internal webserver, listening on port 8765
Feb 07 16:51:57 CKS-Worker falco[30809]: 16:51:57.772908000: Notice Privileged container started (user=root user_loginuid=0 command=container:
Feb 07 16:51:57 CKS-Worker falco[30809]: 16:51:57.777066000: Notice Privileged container started (user=<NA> user_loginuid=0 command=container:
Feb 07 16:51:57 CKS-Worker falco[30809]: 16:51:57.802682000: Notice Privileged container started (user=<NA> user_loginuid=0 command=container:
lines 1-19/19 (END)
```

Falco rules would be located in /etc/falco.

```
root@CKS-Worker:/etc/falco# ls
falco.yaml  falco_rules.local.yaml  falco_rules.yaml  k8s_audit_rules.yaml  rules.available  rules.d
```

Let's give it a test by performing administrative actions, such as writing additional information inside /etc/<whatever file>, inside a Pod (container) and see what logs

```
- kubectl exec test -it -- bash
- echo user >> /etc/passwd
```

Traced Logs (shown in /var/log/syslog. search with keyword “falco”)

Falco Rule Management

All default Falco rules are in /etc/falco/falco_rules.yaml. If administrators would like to check how each rule works, we could use text editor (vim/nano) to view the content.

There are 2 essential ways of changing Falco rules, one by *changing the rule found in falco_rules.yaml directory* and the other by *adding the new rule in falco_rules.local.yaml for overwriting the existing ones*.

Copy the content and append to the bottom of falco_rules.local.yaml

Perform the same action like executing into a terminal within Pod (container) and check the worker node's /var/log/syslog to see what is showing.

Immutability

Immutable basically is a fancy way of saying unchangeable. With this concept in mind, it is much easier to understand the goal of setting up this attribute in K8s Pods. At the moment, startupProbe would be used to achieve the purpose.

startupProbe is designed for detecting legacy application (old applications) to have sufficient time for starting up before it is being monitored by K8s. This probe is invented since readinessProbe nad livenessProbe serve other situations.

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: immutable
  name: immutable
spec:
  containers:
  - image: httpd
    name: immutable
    resources: {}
    startupProbe:
      exec:
        command:
        - sh
        - -c
        - |
          rm /bin/touch
          rm /bin/bash
      initialDelaySeconds: 5
      periodSeconds: 5
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
jonw@CKS-Master:~$ kubectl exec immutable -it -- bash
OCI runtime exec failed: exec failed: container_linux.go:349: starting container process caused "exec: \"bash\": executable file not found in
$PATH": unknown
command terminated with exit code 126
```

Another immutable Pod example with read-only root file system. This time, empty directory needs to be created for writing down the essential logs.

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: immutable2
  name: immutable2
spec:
  containers:
  - image: httpd
    name: immutable2
    resources: {}
    securityContext:
      readOnlyRootFilesystem: true
    volumeMounts:
    - mountPath: /usr/local/apache2/logs
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

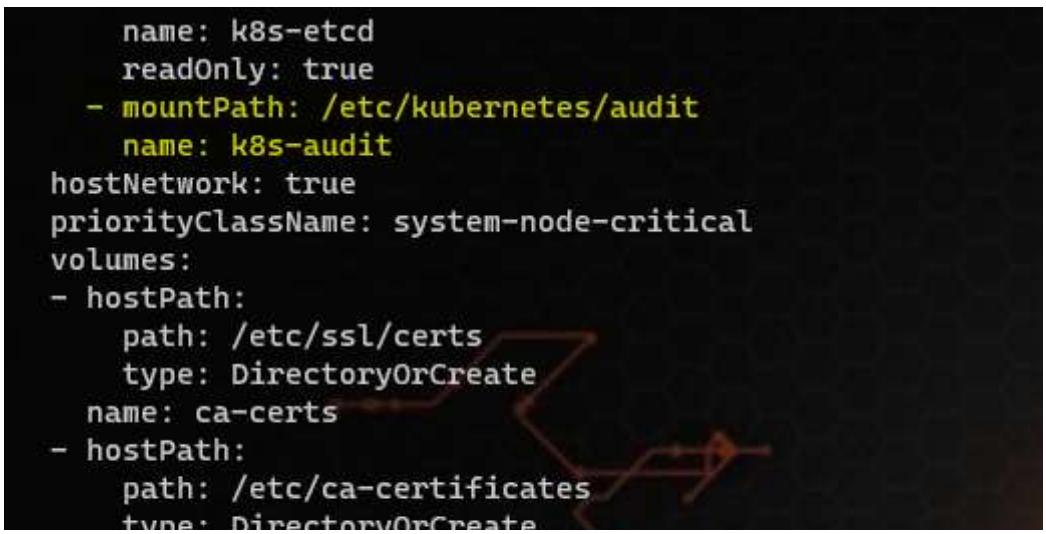
kube-apiserver Auditing

Auditing on kube-apiserver is enabled to understand exactly how every request starts and ends in the K8s cluster. A simple kube-apiserver auditing policy template can be searched here.

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1
kind: Policy
```

After creating a directory under /etc/kubernetes called “audit”, we put the created policy.yaml under the directory /etc/kubernetes/audit. Then, we head over to kube-apiserver configuration file to ensure all configuration files are being loaded. Here is the detailed information. However, the following screenshot could be an easier way enable the same functionality.

Be aware that there is NO “readOnly: true” in k8s-audit volumeMount.



```
      type: DirectoryOrCreate
    name: k8s-certs
  - hostPath:
      path: /usr/local/share/ca-certificates
      type: DirectoryOrCreate
    name: usr-local-share-ca-certificates
  - hostPath:
      path: /usr/share/ca-certificates
      type: DirectoryOrCreate
    name: usr-share-ca-certificates
  - hostPath:
      path: /etc/kubernetes/etcd
      type: DirectoryOrCreate
    name: k8s-etcd
  - hostPath:
      path: /etc/kubernetes/audit
      type: DirectoryOrCreate
    name: k8s-audit
status: {}
```

Checking the logs with

```
sudo tail -f /etc/kubernetes/audit/logs/audit.log
```

Every information starts with “{“ and ends with “}”, in other words, JSON format. We could get a clear visual by putting the content in any online JSON formatter. For example, I would use this site to get formatted JSON data.

```
    "requestURI": "/readyz",
    "verb":"get",
    "user":{
      "username":"system:anonymous",
      "groups":[
        "system:unauthenticated"
      ]
    },
    "sourceIPs":[
      "192.168.1.4"
    ],
    "userAgent":"kube-probe/1.20",
    "responseStatus":{
      "metadata":{

      },
      "code":200
    },
    "requestReceivedTimestamp":"2021-02-06T19:07:40.079594Z",
    "stageTimestamp":"2021-02-06T19:07:40.086859Z",
    "annotations":{
      "authorization.k8s.io/decision":"allow",
      "authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"system:public-info-viewer\" of
    }
}
```


Here is the template of customizing the kube-apiserver auditing rules, so administrators could avoid seeing all the information and only focus on the ones that matter.

```yaml
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage.
omitStages:
  - "RequestReceived"
rules:
  # Log pod changes at RequestResponse level
  - level: RequestResponse
    resources:
    - group: ""
      # Resource "pods" doesn't match requests to any subresource of pods,
      # which is consistent with the RBAC policy.
      resources: ["pods"]
  # Log "pods/log", "pods/status" at Metadata level
  - level: Metadata
    resources:
    - group: ""
      resources: ["pods/log", "pods/status"]

  # Don't log requests to a configmap called "controller-leader"
  - level: None
    resources:
    - group: ""
      resources: ["configmaps"]
```

```
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
    - group: "" # core API group
      resources: ["endpoints", "services"]

  # Don't log authenticated requests to certain non-resource URL paths.
  - level: None
```

AppArmor

There is a lot of articles explaining what AppArmor does, but I would put down how I learn its core concepts and how I think it is applied in day-to-day scenarios. Essentially, AppArmor generates profiles and these profiles controls what processes could be executed and what could not. For example, if administrators write a script about "write" and "delete" a file in a certain directory and use AppArmor to generate a profile around it. By the second time the script is executed, the associated AppArmor rule would be initiated and for most cases, the script would be terminated with some failure until the associated profile is altered otherwise. This is a pretty good article that explains the core concepts of what AppArmor does.

In practice, we could leverage AppArmor profile in K8s Pods, or to be more precise, in containers, to prevent any intentional parties to perform actions other than allowed ones.

We first would need to an AppArmor profile (let's call the customized profile "test") and load it into the nodes that would be hosting the pods, it should be the worker nodes unless the cluster was configured differently.

Load the new AppArmor profile

```
sudo apparmor_parser /etc/apparmor.d/test
```

```
sudo aa-status
```

Test creating a K8s pod with loading this AppArmor profile and see whether the container really is managed by the defined actions. For detailed information, check here.

Test the allowed actions by executing into the terminal of the container.

```
kubectl exec apparmor -it - bash
```

Seccomp

Seccomp is the abbreviation of secure computing mode. To my understading, it basically restricts the instance, virtual machine or container, to perform any action other than the preset ones. That includes exit(), sigreturn(), read() and write(). This is from Linux manual page.

In practice, if we would like to have K8s Pods (containers) to run under this condition, we would need to make sure this configuration (let's call the configuration "default.json") is loaded somewhere in the core services. In this case, that would be in kubelet. The default path to read seccomp profile for kubelet is /var/lib/kubelet.

```
cat /var/lib/kubelet/seccomp/<seccomp_profile.jsono
```

Here is the way to setup seccomp in a K8s Pod(container).

Test the allowed actions by executing into the terminal of the container.

```
kubectl exec seccomp -it - bash
```

Linux Basic Administration

It feels like it is the best to prepare this section of the CKS exam through some other existing online learning material. For example, Digital Ocean provides a thorough

This last part of the CKS preparation series is extremely broad and it is closely related with many common Linux administration concepts. If you have not much experience around Linux, please make sure to do extra research on the Net and do plenty of practice before going for the real exam as the testing environment would only have more restrictions. This article covers the last piece of what CKS would be testing and the next article would just be giving some exam hacks and advice! Happy Learning!

Kubernetes Security Preparation Certification Falco Syscalls