Building and Validating Kubernetes Detection Capability with Leonidas

Workshop Guide

DEATHcon 2024

WITH secure

Document History

Date	Change	Description	Issue
2024-10-28	Document Published	Original Version	1.0

Table of Contents

1 Workshop Guide 4

1.1 Preparing the Environment 5

Install Kubernetes 5

Set Up Monitoring 7

Integrating with a SIEM 15

1.2 Simulating Kubernetes Attacks with Leonidas 31

Deploy Leonidas 32

Execute Test Cases 37

Detection Engineering 40

1.3 Emulating Real-World Adversaries 43

Leonidas & Jupyter Notebooks 43

SSWW Run Trough 45

Validating Detection Capability 50

2 Exercise – RBAC Buster 52

3 Closing Notes 53

Appendix I - Troubleshooting 54

Issue: Minikube API Server Failed to Start? 54

Issue: Leonidas Pod Failed to Start? 55

1 Workshop Guide

Welcome to the “Building and Validating Kubernetes Detection Capability with Leonidas” workshop of DEATHcon24! This guide will walk you step-by-step through the content we will cover, and serves as written reference for use alongside the video. It is structured as follow:

```
1 Main Section
  1.1 Chapter A
    Part I
      1. Step X
         instructions to complete it ...
      2. Step Y
         instructions to complete it ...
    Part II
      1. Step Z
         ...
```

As a registered attendee, you should be provided with the following materials:

- This guide
- An archived folder `student-materials/`
- Access to the DEATHcon Lab’s Elastic deployment “elastic-deathcon2024”

  Kibana Link: https://elastic-deathcon2024.kb.eastus.azure.elastic-cloud.com:9243/

  Your Elastic user will also need to have the “Admin” role for all deployments

Additionally, you will need to “bring your own”:

- Linux VM with a desktop environment.

  There is no need join this VM to the DEATHcon Lab, so you can run any virtualisation technology and Linux flavour you want. Note however that the content has been recorded and tested successfully for Ubuntu 22.04, which is what this course recommends.

  Kubernetes clusters are notoriously resource intensive, so significant resources might be needed. We recommend providing your VM at least

  - 8GB of RAM
  - 4 CPU Processors
  - 30 GB of disk space

Once you confirm all is in place, log on to your VM and let’s get started!

1.1 Preparing the Environment

In our box, we will start a local Kubernetes cluster & deploy some test resources in it. Before doing so, however, let's install some utilities that will come in handy later.

1. Install Helper utilities

```
$ sudo apt install vim net-tools curl
```

Install Kubernetes

Next up, our Kubernetes environment (aka K8S). Real-world Kubernetes clusters consist of several hosts ("nodes"), and enrolling them under a single orchestration plane ("bootstrapping") is no trivial task. To simplify this process here, and abstract away this hardware complexity using only a single machine, we will use the minikube tool – to spin up the smallest possible, 1-node cluster.

2. Install Minikube

Following the Minikube Quickstart guide, let's download the binary and install it on our system

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

We should now have the minikube binary in our path:

```
$ minikube version
minikube version: v1.34.0
commit: 210b148df93a80eb872ecbeb7e35281b3c582c61
```

Minikube can leverage a range of different backends to spin up the virtual nodes upon, or "drivers". As the simplest possible option for our given setup, we will use the Docker driver, to run our single node as a container.

This might get a bit confusing, so it's worth clarifying: As we all know, Kubernetes is a *container orchestration* solution, so the workloads are running as containers. In our case, to abstract away the cluster hardware layer, we'll also use a container as the node. Effectively, that means we will end up with a Docker on Docker[1] environment, or nested container condition.

So let's proceed to install Docker on our VM, to enable minikube's docker driver.

3. Install Docker Engine on VM

Following the instructions here, we will install Docker engine through the `apt` repository

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl
$ sudo install -m 0755 -d /etc/apt/keyrings
```

[1] Note: To run containers on the nodes, Kubernetes needs a "container runtime". One of these supported runtimes used to be Docker – or to be accurate, `dockershim` – a shim allowing the node's Docker to be used as the Kubernetes runtime. However, as of version 1.24, Kubernetes no longer supports `dockershim`. Docker can still be used, just with an alternative shim: `cri-dockerd`, and this will be the implementation that our Minikube cluster will default to, hence the first "Docker" of Docker on Docker.

```
$ sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
$ sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

To confirm Docker Engine has been installed successfully:

```
$ sudo docker run hello-world
...

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

For a final post-installation step, let's add our user to the `docker` group too, for easier interaction with the docker runtime:

```
$ sudo usermod -aG docker $USER
```

4. Bootstrap a Kubernetes cluster

Now we can start Minikube, specifying docker as our driver.

```
$ minikube start --driver=docker
```

To confirm our self-contained single-node cluster is up and running, we will list the pods across all namespaces. The first time we interact with Kubernetes using kubectl, minikube will download the appropriate version of the kubectl utility for us before executing the command:

```
$ minikube kubectl -- get pods -A
    > kubectl.sha256:  64 B / 64 B [-------------------------] 100.00% ? p/s 0s
    > kubectl:  53.77 MiB / 53.77 MiB [------------] 100.00% 34.43 MiB p/s 1.8s
NAMESPACE     NAME                               READY   STATUS    RESTARTS      AGE
kube-system   coredns-6f6b679f8f-q5pb9           1/1     Running   0             31s
kube-system   etcd-minikube                      1/1     Running   0             37s
kube-system   kube-apiserver-minikube            1/1     Running   0             37s
kube-system   kube-controller-manager-minikube   1/1     Running   0             37s
kube-system   kube-proxy-xd46n                   1/1     Running   0             31s
kube-system   kube-scheduler-minikube            1/1     Running   0             37s
kube-system   storage-provisioner                1/1     Running   1 (10s ago)   35s
```

Optionally, you can also set an alias for the downloaded binary for convenience:

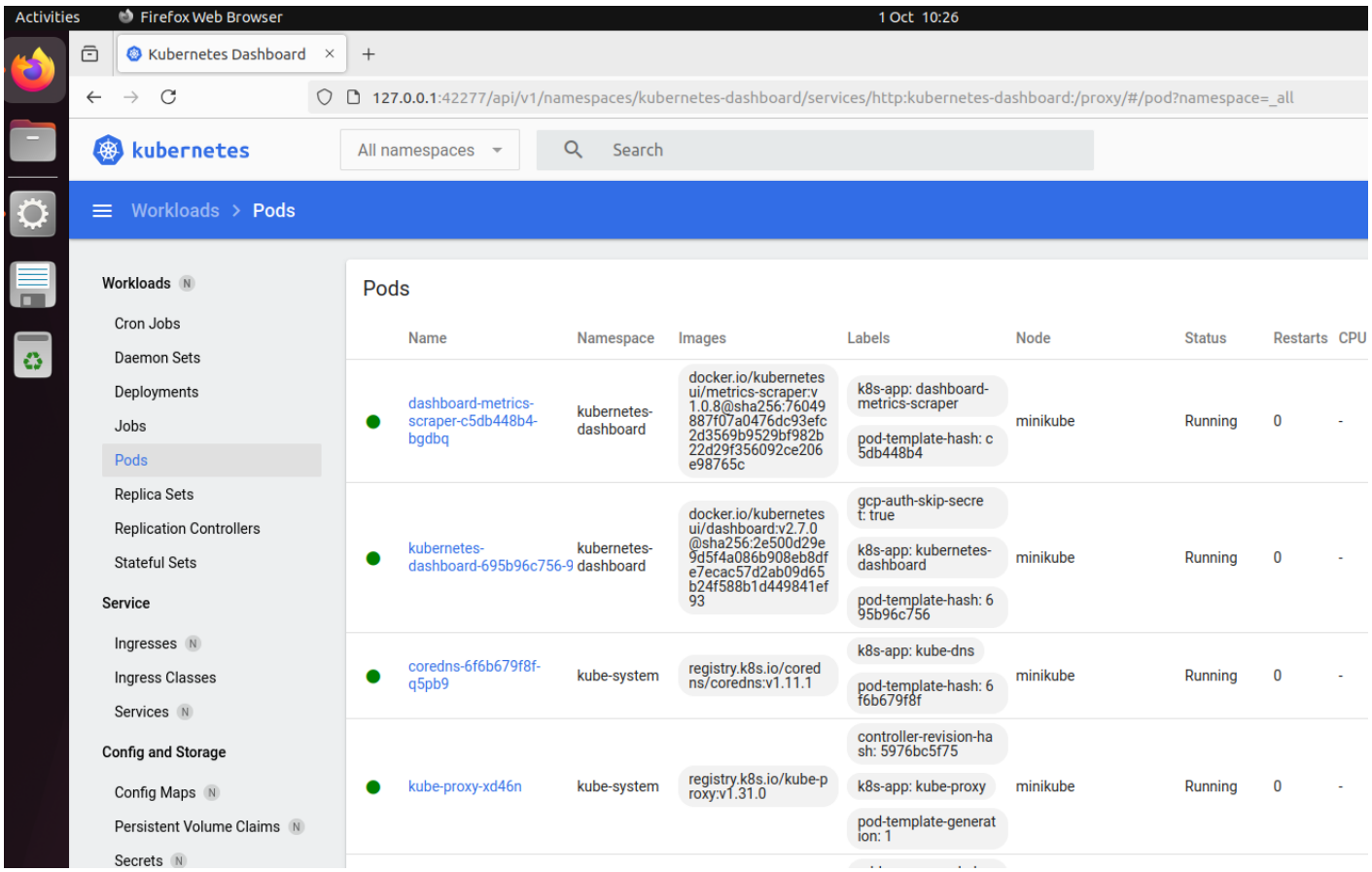
```
$ alias kl=`find / -name kubectl -executable 2>/dev/null`
```

5. (Optional) install Kubernetes Dashboard

Another optional step to familiarise with our new cluster, especially for those new to Kubernetes, is to explore the resources through a GUI, instead of interacting with kubectl. This can be achieved with the Kubernetes dashboard, conveniently integrated into minikube as an addon.

```
$ minikube dashboard
```

This will spawn a new Firefox tab pointing at the dashboard, authenticated and ready:



Our cluster is now completely empty. And although a default Kubernetes installation includes dozens of default resources, let's populate it with some sandbox data to play around with:

6. Create some test data

In our `student-materials/` directory, you will find a file called `persepolis-env.yml`, which includes a few resources as well as a "namespace" where they will all be deployed into. Have a read through the concatenation of the YAML definitions and create them once ready:

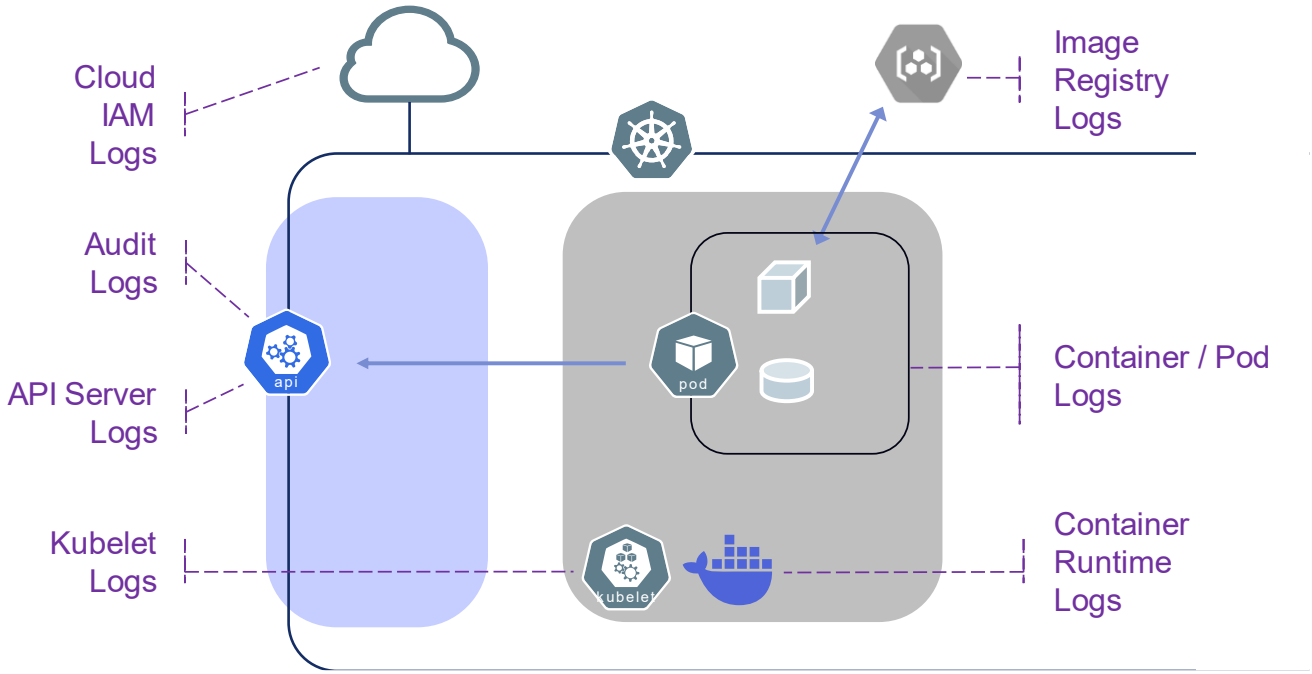
```
$ kubectl apply -f persepolis-env.yml
namespace/persepolis created
serviceaccount/pyramid-god created
secret/revolution-dna created
pod/netutils created
```

Set Up Monitoring

With a cluster up and running, we can now look into how we can monitor activity in it.

Kubernetes Log Sources

The most important building block, when it comes to monitoring, is arguably Log Sources. If you have a look at the official Kubernetes documentation page about logging, you'll find a typical cluster would offer plenty of log sources...



Of all these log sources, only a few are actually useful for security, and of those few the most valuable is admittedly the API server's Audit Logs.

To monitor control plane activity, we need to define an Audit Policy, and enable Audit logging.

1. Define an Audit policy

Depending on configuration, Kubernetes Audit logs can be extremely verbose and loud, leading to gigabytes of space required. We still need to have sufficient detail, however, for writing alert rules and detecting attacks. Finding just the right balance for your environment will be a journey - it will require baselining what's normal and what isn't, as well as continuous tuning and verification. Luckily, Kubernetes auditing controls are very configurable, providing fine granularity on the verbosity for each different event.

A good policy to get started with, is that set for GCE clusters, built by the script found in the following link: github.com/kubernetes/kubernetes/blob/master/cluster/gce/gci/configure-helper.sh

We will also use this example policy, it's provided as `audit-policy.yaml` in the student materials directory. Feel free to take a moment to give this a read.

2. Enable Audit Logging

With our policy ready, we now need to point the API server to it, and re-start it with the extra command line flags necessary. Unfortunately, this is not so trivial in minikube due to the simplification conventions it uses. We need to use two workarounds:

- To pass the Audit Policy from the VM -> into the node -> and from there into the kube-apiserver container, we can use a documented workaround that leverages the SSL certs directory. This involves dropping the file in a directory that will be synced into the container, and restarting minikube with the necessary flags to be passed to the API server.

```
$ mkdir -p ~/.minikube/files/etc/ssl/certs
$ cp audit-policy.yaml ~/.minikube/files/etc/ssl/certs/

$ minikube stop

$ minikube start \
  --extra-config=apiserver.audit-policy-file=/etc/ssl/certs/audit-policy.yaml \
  --extra-config=apiserver.audit-log-path=/var/log/audit.log
```

- And to persist the audit logs from within kube-apiserver pod > out onto the node, we will make some modifications to minikube's default manifest for the API server. This will introduce an extra "hostPath" volumeMount and its respective volume, mapping to the node filesystem.

  First, we get a shell on the minikube node, and install a code editor

```
$ minikube ssh
docker@minikube# sudo su
root@minikube# sudo apt update
root@minikube# sudo apt install vim
```

  Then we edit the API server manifest to add the parts highlighted below:

```
root@minikube# cat /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  ...
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.49.2
    ...
    - --audit-log-path=/var/log/audit.log
    - --audit-policy-file=/etc/ssl/certs/audit-policy.yaml
    ...
    volumeMounts:
    - mountPath: /etc/ssl/certs
      name: ca-certs
      readOnly: true
    ...
    - mountPath: /usr/share/ca-certificates
      name: usr-share-ca-certificates
      readOnly: true
    - mountPath: /var/log
      name: var-log
      readOnly: false
  hostNetwork: true
  ...
  volumes:
  - hostPath:
      path: /etc/ssl/certs
      type: DirectoryOrCreate
    name: ca-certs
  ...
```

```
  - hostPath:
      path: /usr/share/ca-certificates
      type: DirectoryOrCreate
    name: usr-share-ca-certificates
  - hostPath:
      path: /var/log
      type: DirectoryOrCreate
    name: var-log
status: {}
```

These changes are provided in `audit.patch` file in student materials. This can be used to speed up their re-application e.g. in case of a cluster redeployment:

```
$ patch kube-apiserver.yaml audit.patch
```

Kubernetes Attack Surfaces

By collecting API server audit events, we are covering attacks against the orchestration layer of Kubernetes - the control plane itself.

However, an attack surface far more explored (and exploited, as confirmed by existing threat intelligence) is that of containers themselves. This involves vulnerabilities introduced by the workloads, such as web applications, or backdoored images with piggyback shell commands to do malicious stuff, alongside the intended workload.

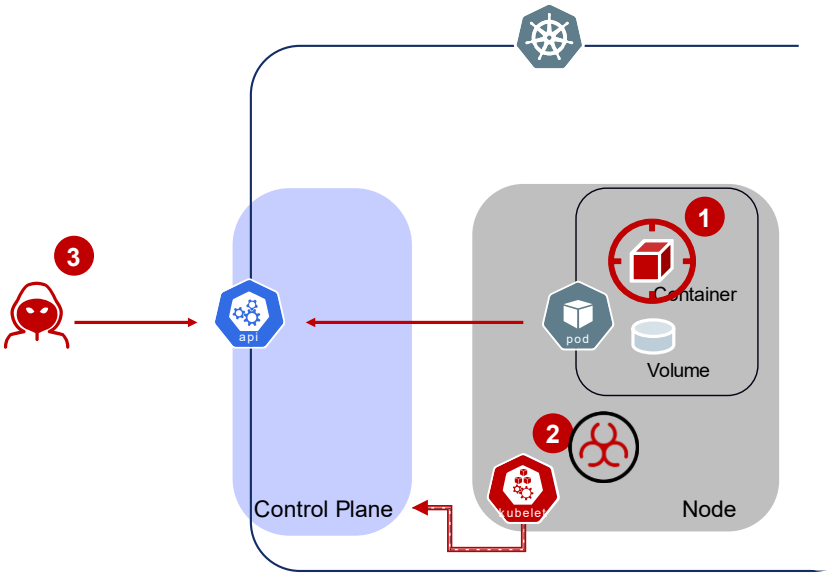
And what about the hosts where containers are running upon? Is it worth monitoring the nodes? Does their risk profile change when they're enrolled into a Kubernetes cluster, or shall we approach them as standard Linux / Windows servers requiring hardening? A common target, for example, is the Kubelet, which also talks to the control plane with elevated privileges…

Broadly speaking, we can identify three different levels where attacks can originate from, when it comes to Kubernetes threat modelling:

1. Container Level

2. Node Level

3. Control Plane Level

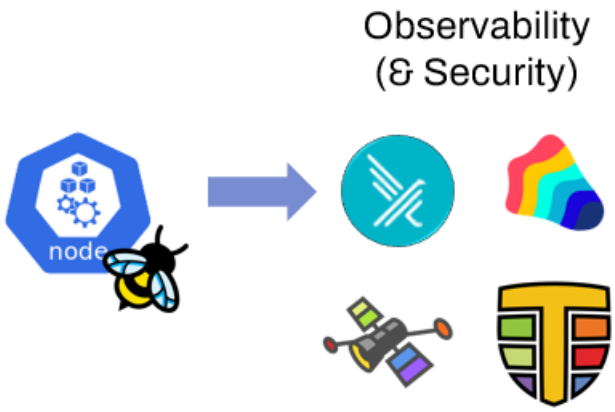


In the previous part we covered the control plane, so to introduce some visibility into the other two attack surfaces as well, we'll install the Falco security monitoring solution, at the host level

3. Install Falco

Container / Node-level Monitoring

Security monitoring and system observability on cloud-native systems is not a new concept. Plenty of research has been carried out on EDR-style monitoring capabilities for Linux systems, leveraging the eBPF kernel functionality. This has resulted in various Open Source solutions like Tetragon, Tracee, Cilium Hubble and Sysdig's Falco, with most of them easy to integrate in Kubernetes environments.



We can install falco with many different ways, and in different locations.

Reading the installation page of documentation, you might be tempted to go for the Kubernetes option. However, this is not the most secure option as deploying it as Kubernetes resources makes it liable to a breach of the orchestration layer. A more resilient alternative is to deploy Falco on nodes directly - e.g. through packages - independent of Kubernetes, to ensure separation in case of control plane compromise.

Installing it on the node would also be the most realistic option – every Kubernetes node running falco as a daemon, monitoring their own containers. For the sake of simplicity and limitations on resources however, we'll install in on our Ubuntu hosts in this workshop. Running nested containers means that despite the extra layer, Kubernetes workloads will still proxy syscalls to the host's kernel, and can thus be hooked by Falco's eBPF module.

All this, is simply a mater of installing the falco apt package:

```
$ curl -fsSL https://falco.org/repo/falcosecurity-packages.asc | \
sudo gpg --dearmor -o /usr/share/keyrings/falco-archive-keyring.gpg
$ echo "deb [signed-by=/usr/share/keyrings/falco-archive-keyring.gpg]
https://download.falco.org/packages/deb stable main" | \
sudo tee -a /etc/apt/sources.list.d/falcosecurity.list
$ sudo apt-get update -y
$ sudo apt-get install -y falco
```

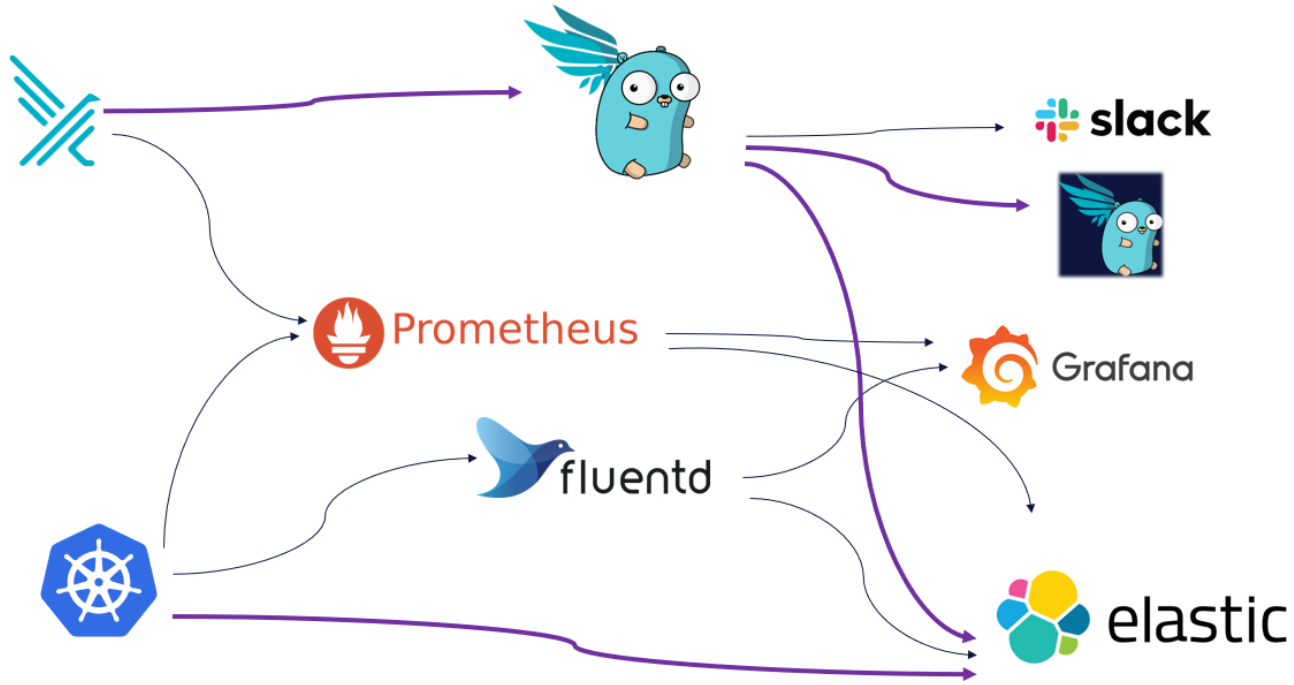
After this, the falco service should be running without any errors:

```
ubuntu@studentvm:~$ sudo systemctl status falco
● falco-modern-bpf.service - Falco: Container Native Runtime Security with modern ebpf
     Loaded: loaded (/lib/systemd/system/falco-modern-bpf.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2024-10-01 13:49:46 BST; 5min ago
```

4. Install falcosidekick and falcosidekick UI

Log Collection Topologies

Natively, falco can send alerts to a limited number of “output channels” such as standard output, files, syslog or a gRPC/HTTP(S) API. To integrate with more systems and commonly used tools, the falcosidekick forwarding daemon was created. falcosidekick can be configured to receive alerts from Falco, and forward them to more than 60 services including Slack, or Discord. Amongst these, falcosidekick can be configured to send alerts to falcosidekick UI, a simple Web UI for displaying latest events. This is only one of the various different log collection setups that one can implement, given the extensible and pluggable nature of Kubernetes. Very popular technologies such as Prometheus or Fluentd can be used to collect logs from nodes, and forward them to a SIEM or other visualisation platforms such as Grafana, for performance monitoring.



We will install falcosidekick and falcosidekick UI as docker containers for simplicity, along with the redis database required.

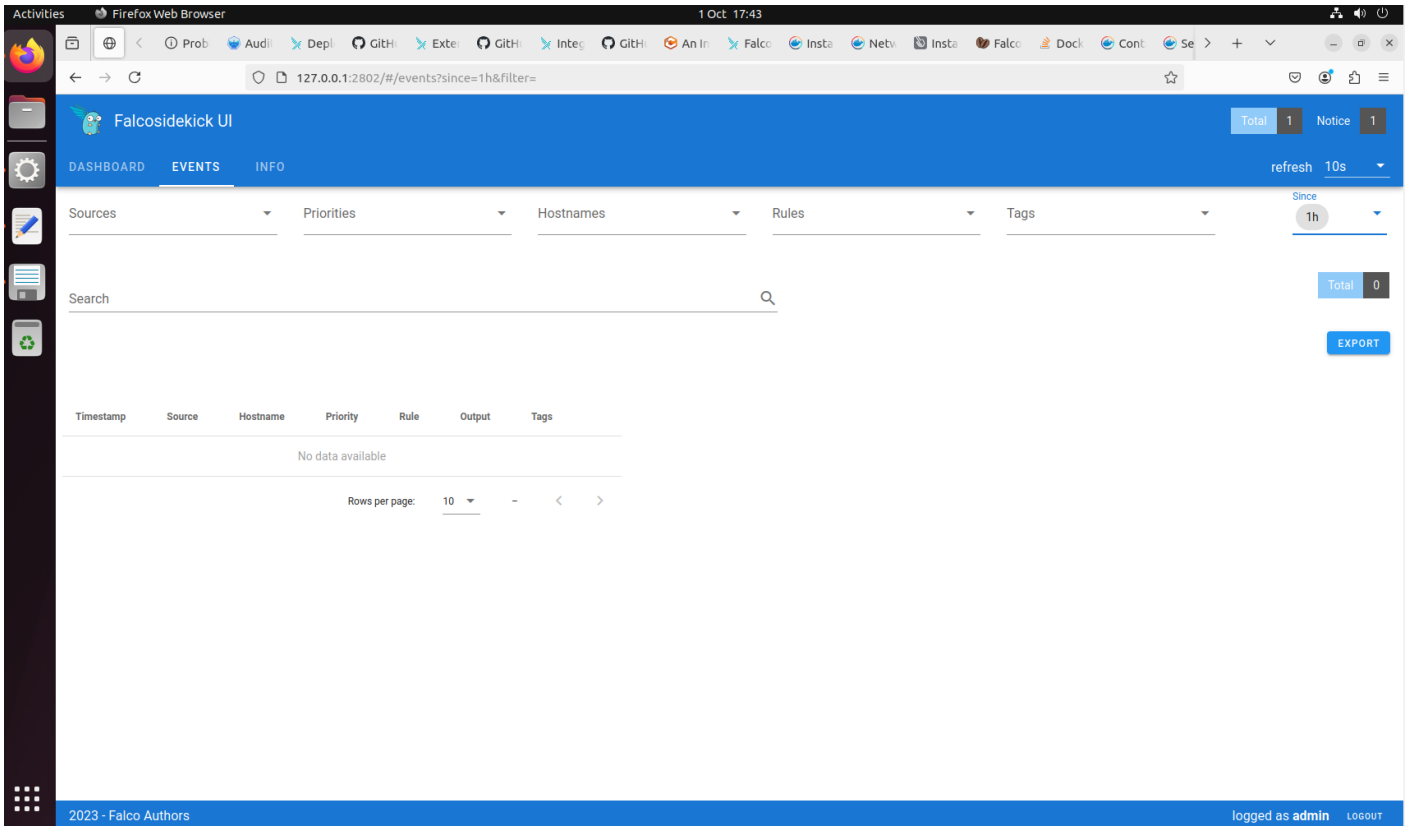
Navigating into the directory where our `docker-compose.yml` is stored, we start the services:

```
$ docker compose up --detach
```

Now, we need reconfigure falco to send data to falcosidekick, and restart the falco service.

```
$ sudo cp falco-to-sidekick.yaml /etc/falco/config.d
$ sudo service falco restart
```

And this will mean that the UI is up and running, on port 2802 of the host, forwarded from the falcosidekick UI container (use default credentials `admin:admin` for the UI, as per the documentation):



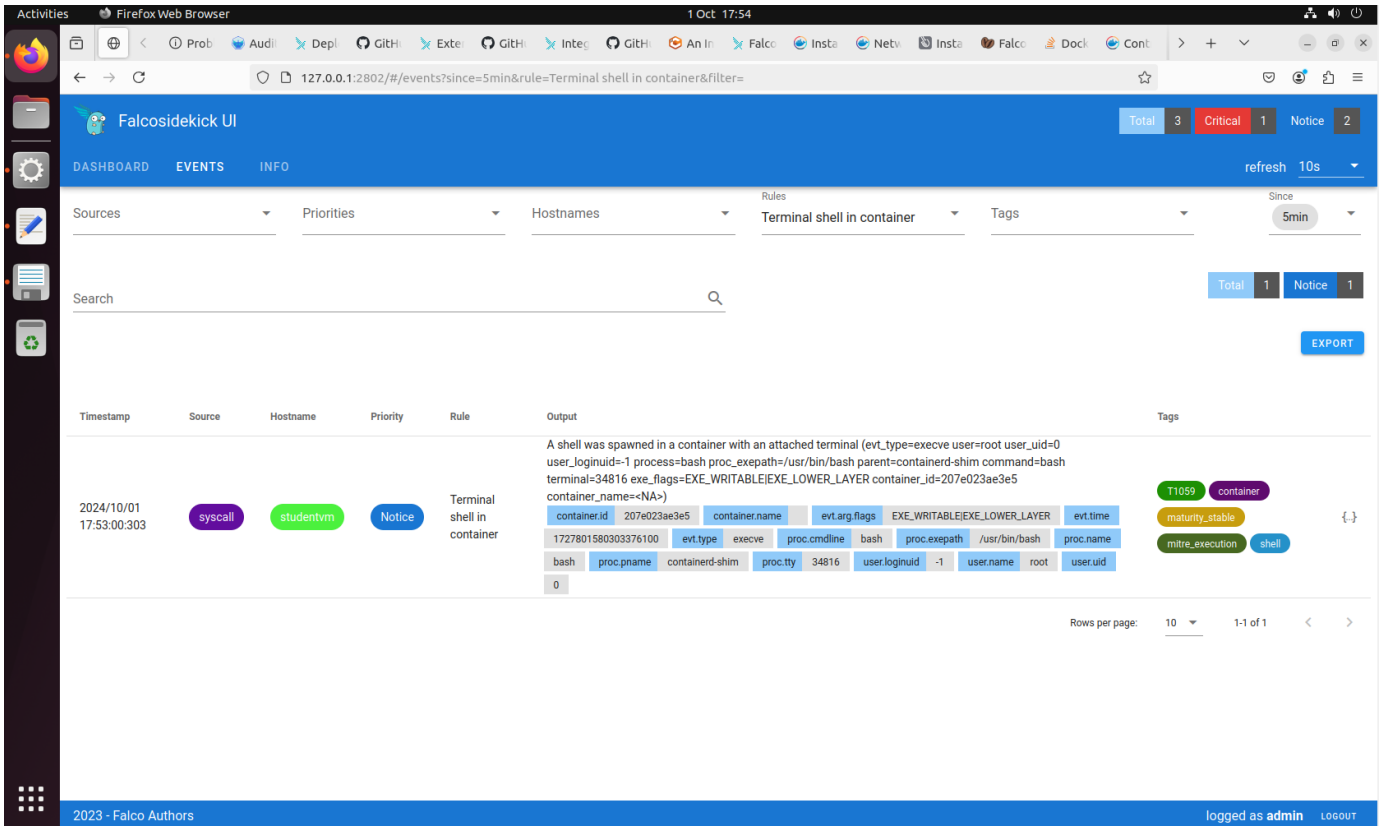
5. Confirm Falco Monitors Kubernetes Containers

Falco is up and running on the host, but is it actually monitoring Kubernetes workloads? To test this, we'll trigger one of Falco's default alert rules: "Terminal shell in container" which detects users getting an interactive shell in a container – an operation unusual in production clusters.

Deploying a test pod from the `netutils-pod.yaml` manifest provided, and shell'ing into it…

```
$ kl apply -f netutils-pod.yaml
pod/netutils created
$ kl exec netutils -ti – bash
```

…should lead to a new entry in the UI – confirming falco's syscall monitoring capabilities:



Integrating with a SIEM

Collecting Kubernetes logs into a SIEM is not a novel challenge. Various solutions exist that enable ingestion of all possible types from a Kubernetes cluster, into a central platform for long-term storage, analysis, visualisation, and monitoring purposes. All of these solutions typically involve an agent tool, that needs to be deployed as Kubernetes resources into your cluster (usually a DaemonSet, which enables execution on every node), which then connects over the internet to your SIEM's URL, authenticating via tokens or credentials.

Elastic, for example, provides a Kubernetes integration for this purpose, which is the technology we will use In this workshop However if your organisation's SIEM of choice is Splunk, have a look at the Splunk Open Telemetry Collector for Kubernetes, for equivalent functionality.

The Elastic Kubernetes Integration focuses primarily on "Observability", but can also be used to easily collect Audit logs from our cluster.

1. Connect your cluster with the lab's Elastic instance

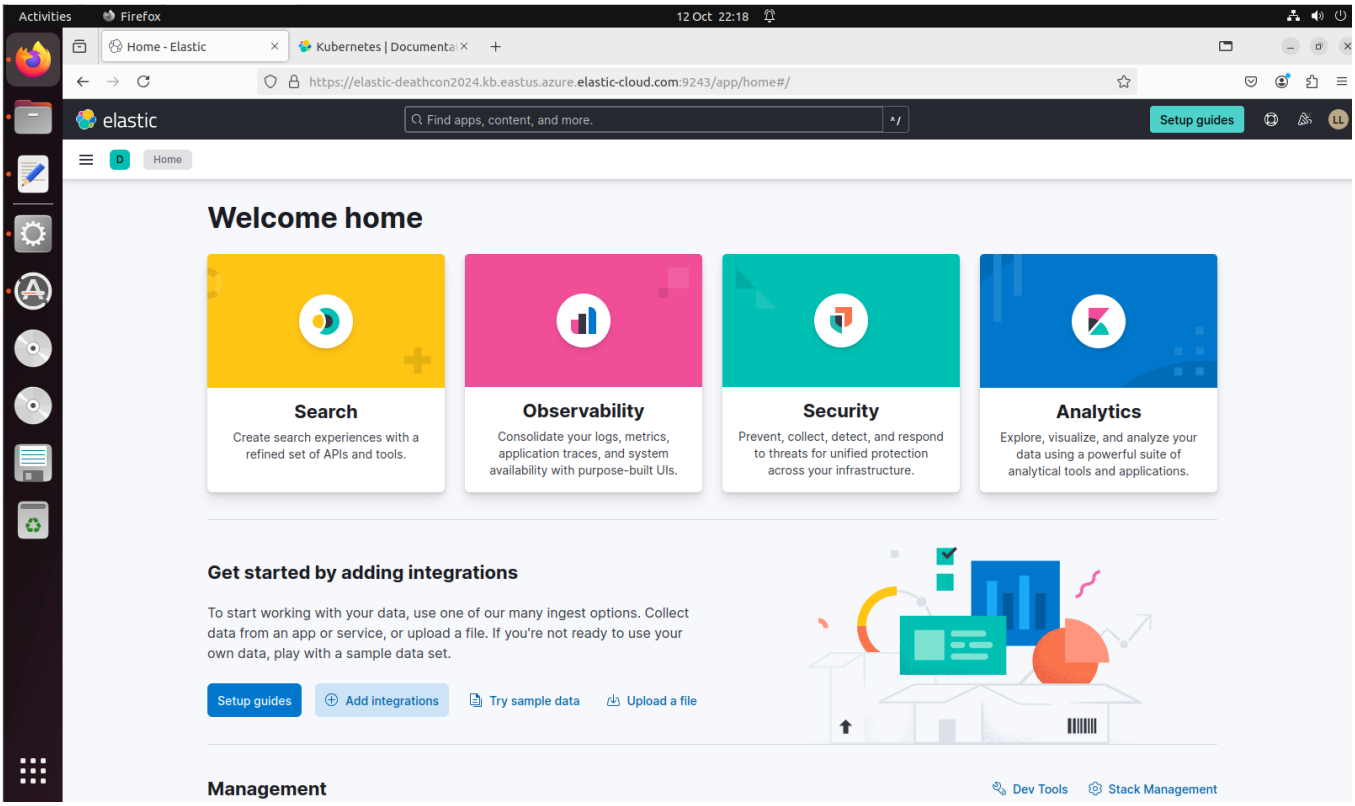
Open the Kibana UI of your Elastic deployment
https://elastic-deathcon2024.kb.eastus.azure.elastic-cloud.com:9243/

If you're not already logged into Elastic, this will redirect you to the login page.

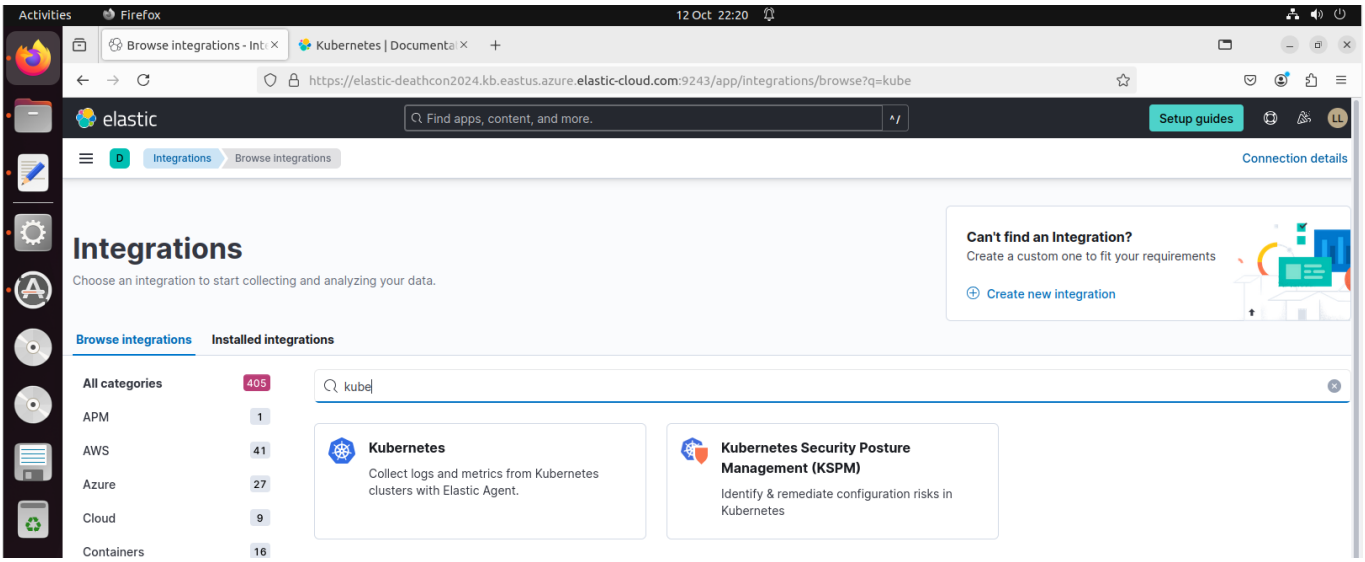
Click the "Elastic Cloud" option, and then the "Microsoft" button, to use SSO.

Once you're logged in, you might also have to be invited to the DEATHLab Elastic Organization if you haven't before. Ask the DEATHcon folks and they'll invite you from the Elastic portal, which will result in an invite link arriving over email (https://mx.deathmail.net).

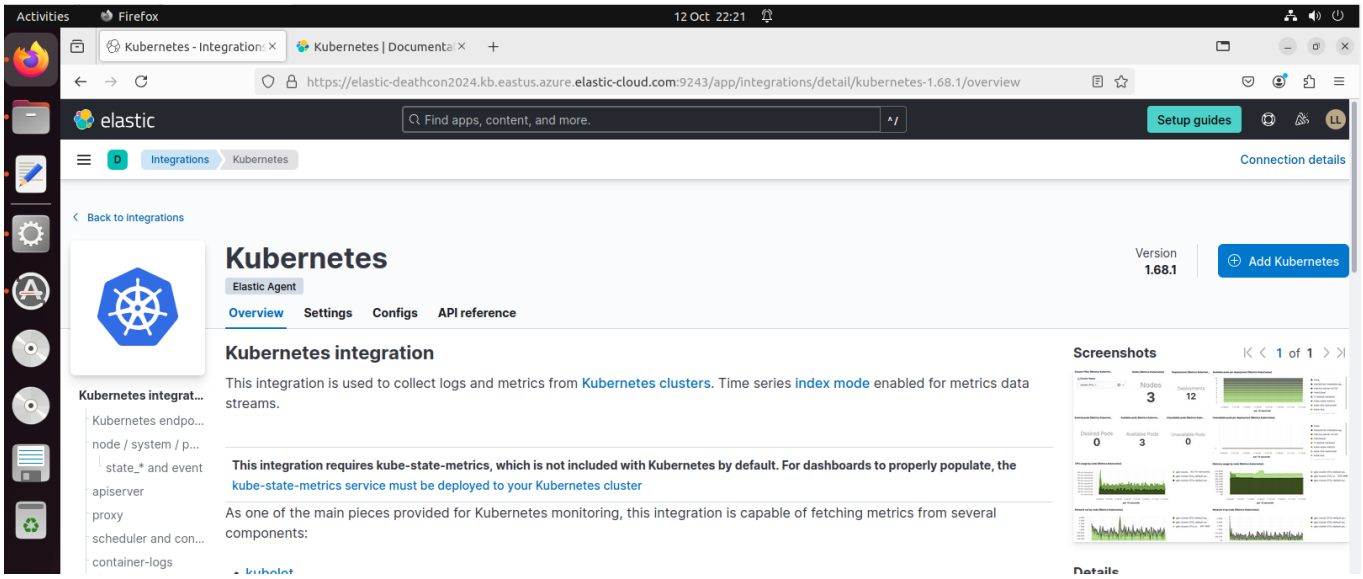
Once you're eventually added to the Organisation, the Kibana interface for the deployment should be reachable.



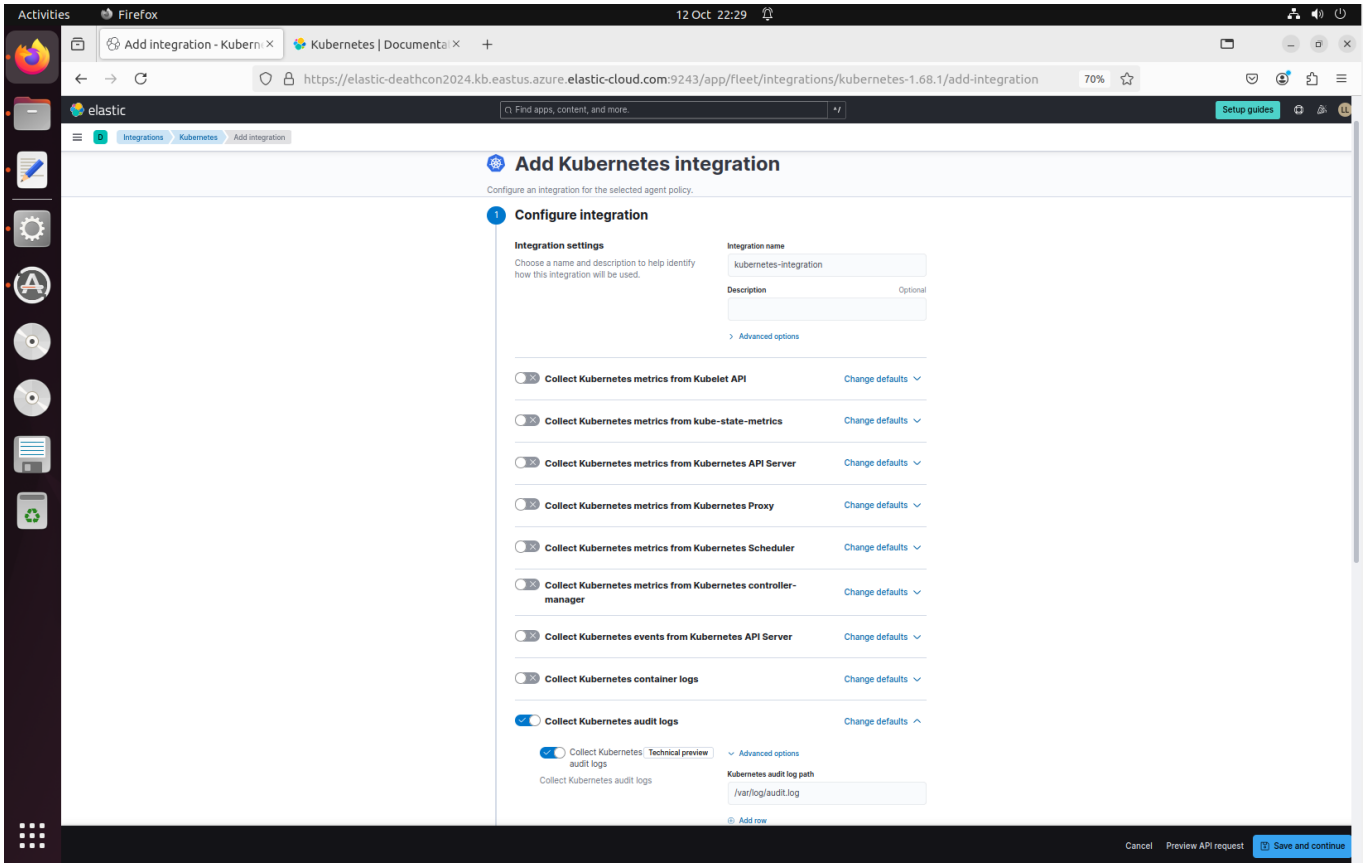
Click the smaller "Add Integrations" button, and search for "Kubernetes"



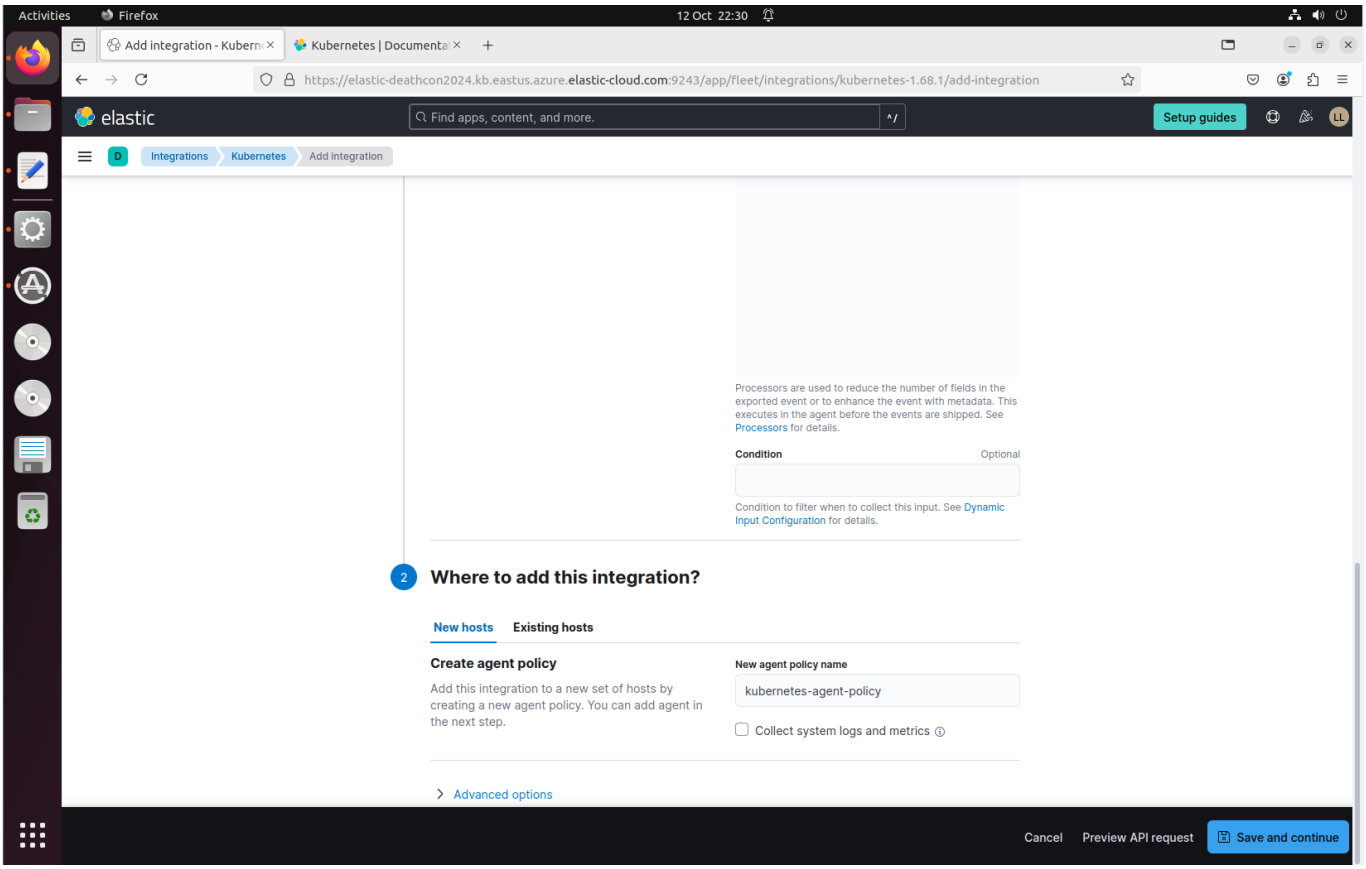
Select the first option, and then "Add Kubernetes" from the top right:



In the Integration settings that will show up, set them all as below:

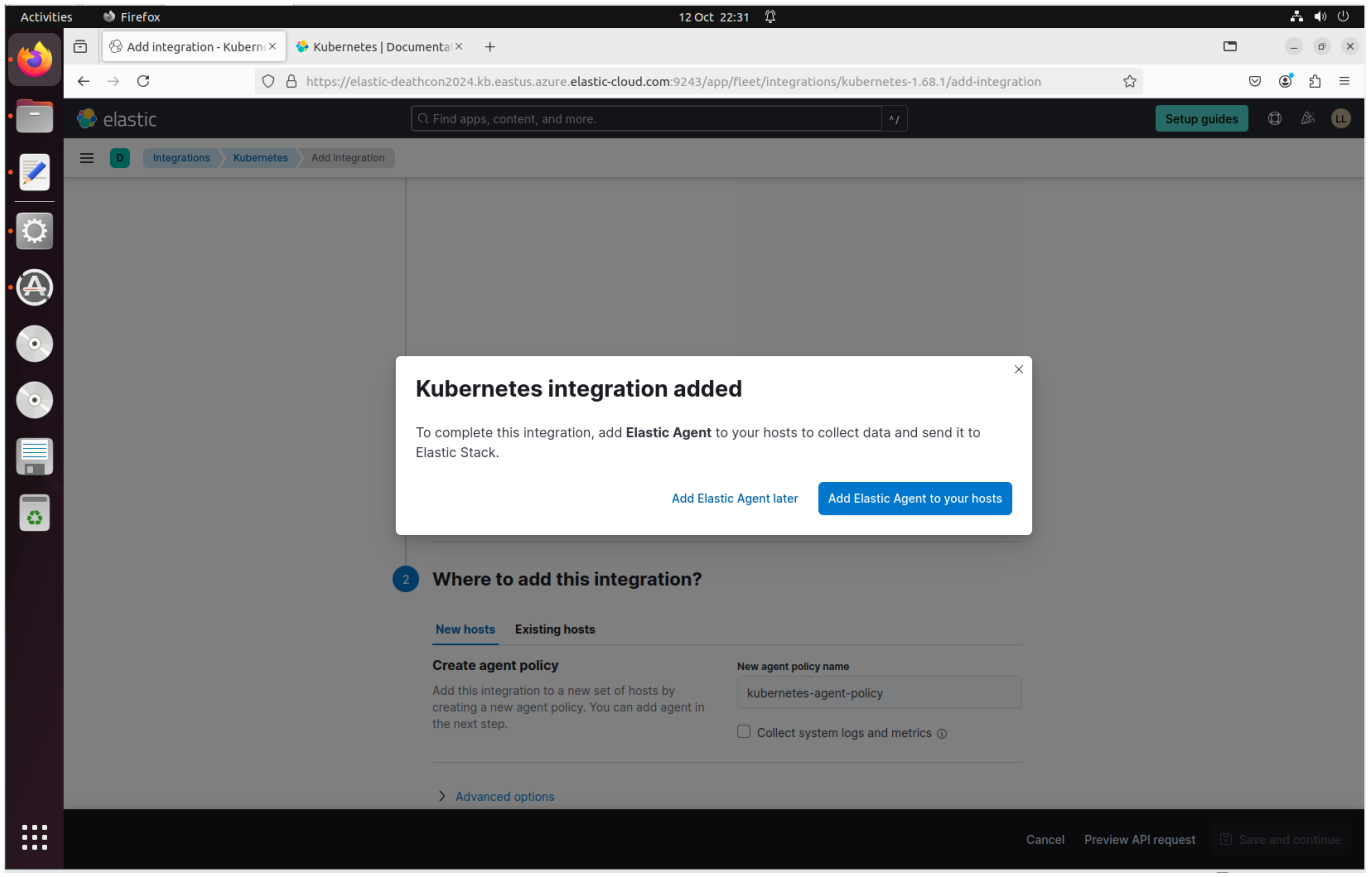


Make sure you also set the audit log path to the volumeMount you configured previously!
`/var/log/audit.log`

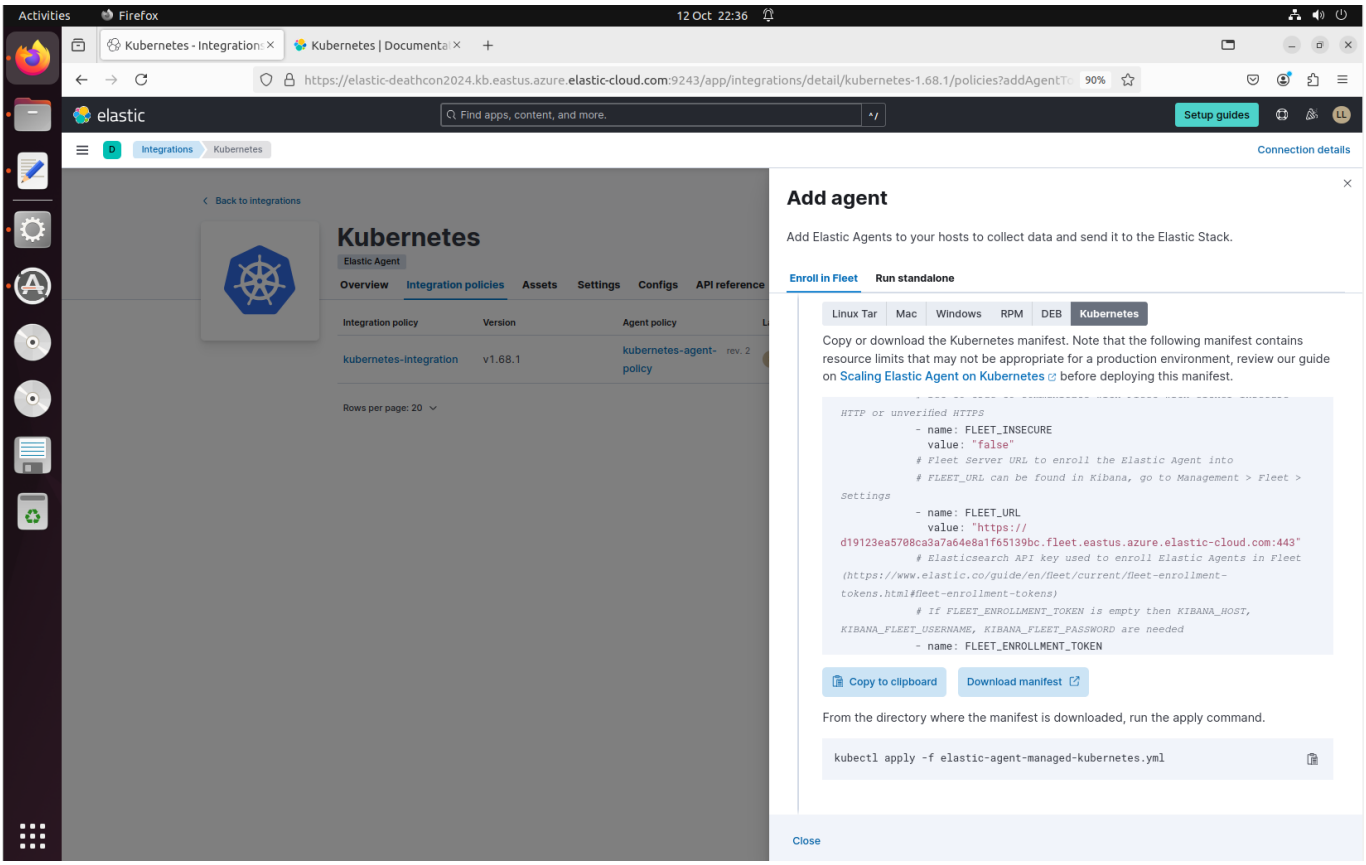


Note that you might have to add a qualifier to the agent policy, as the main one already exists. So name it, for example `kubernetes-agent-policy-<studentID>` where studentID is an identifier unique to you, such as your username

Then, once you hit “Save and continue”, the dialog will appear where you can proceed immediately to “Add Elastic Agents to your hosts”

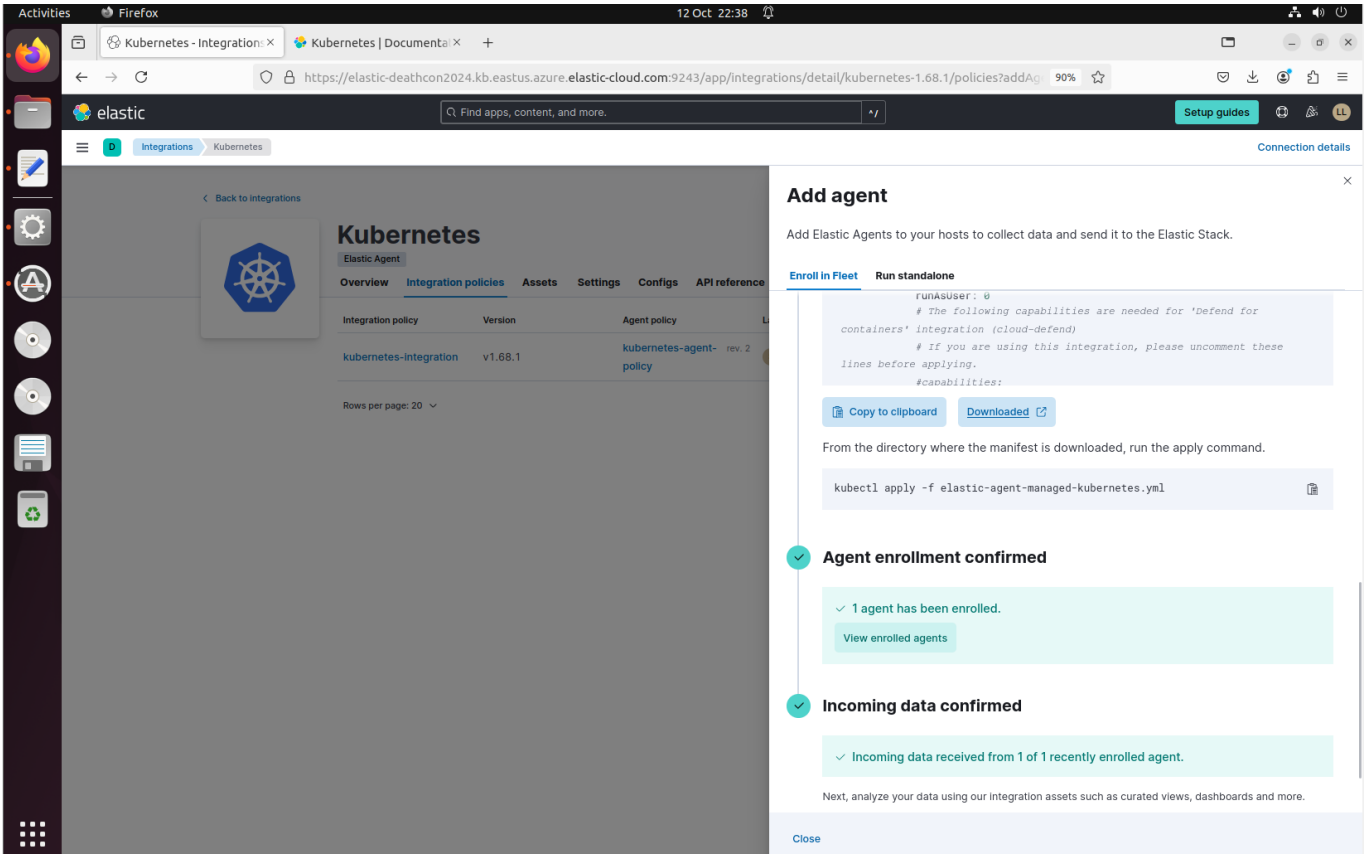


Then on the right sidebar, scroll down to the Kubernetes manifest for the enrolment. This is the YAML manifest of the resources (primarily, the Elastic Agent DaemonSet) that you'll have to deploy inside your cluster, in order for the logs to be collected. It also includes the credentials needed to authenticate to the lab's deployment:



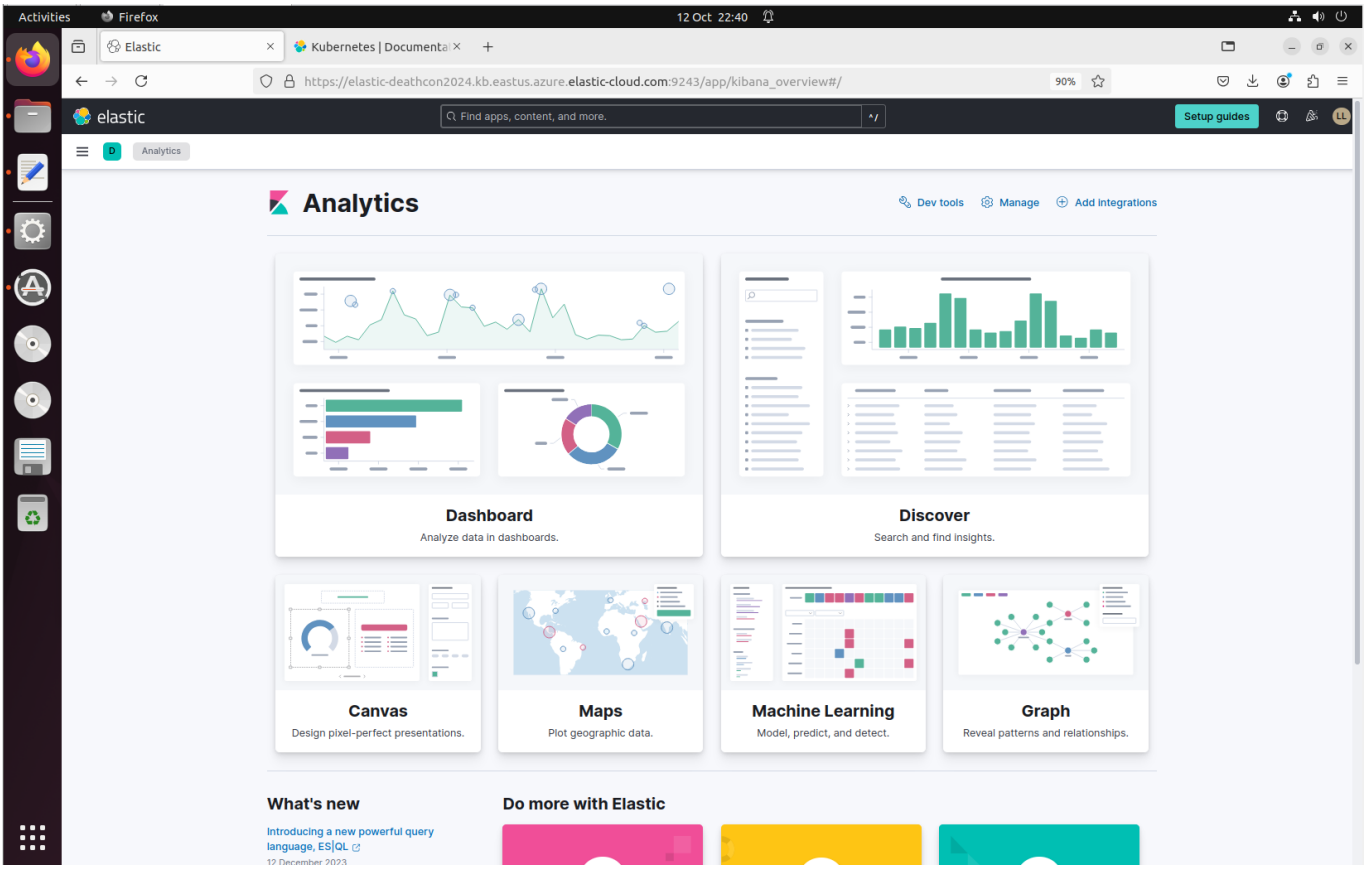
As instructed, download this locally, and run the `kubectl apply -f` command

After a few seconds, the “Listening for agents” spinner will be replaced by one new instance!

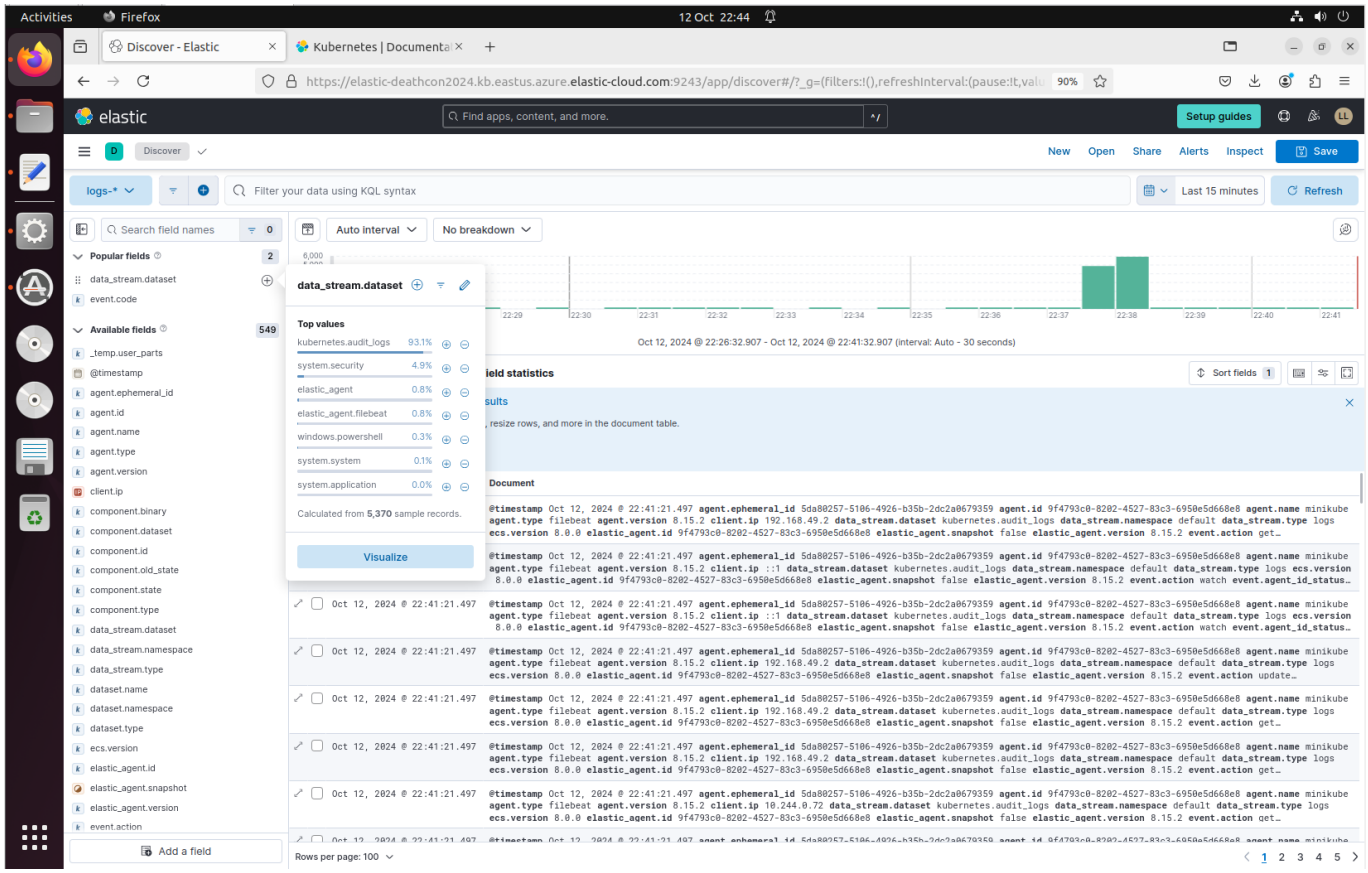


Hit “Close” and navigate back to the Deployment’s home page.

This time, click the large “Analytics” button, and then “Discover”:



There, the `logs-*` Data View should greet you with your cluster’s logs, tagged with a `data_stream.dataset` value of `kubernetes.audit_logs`:

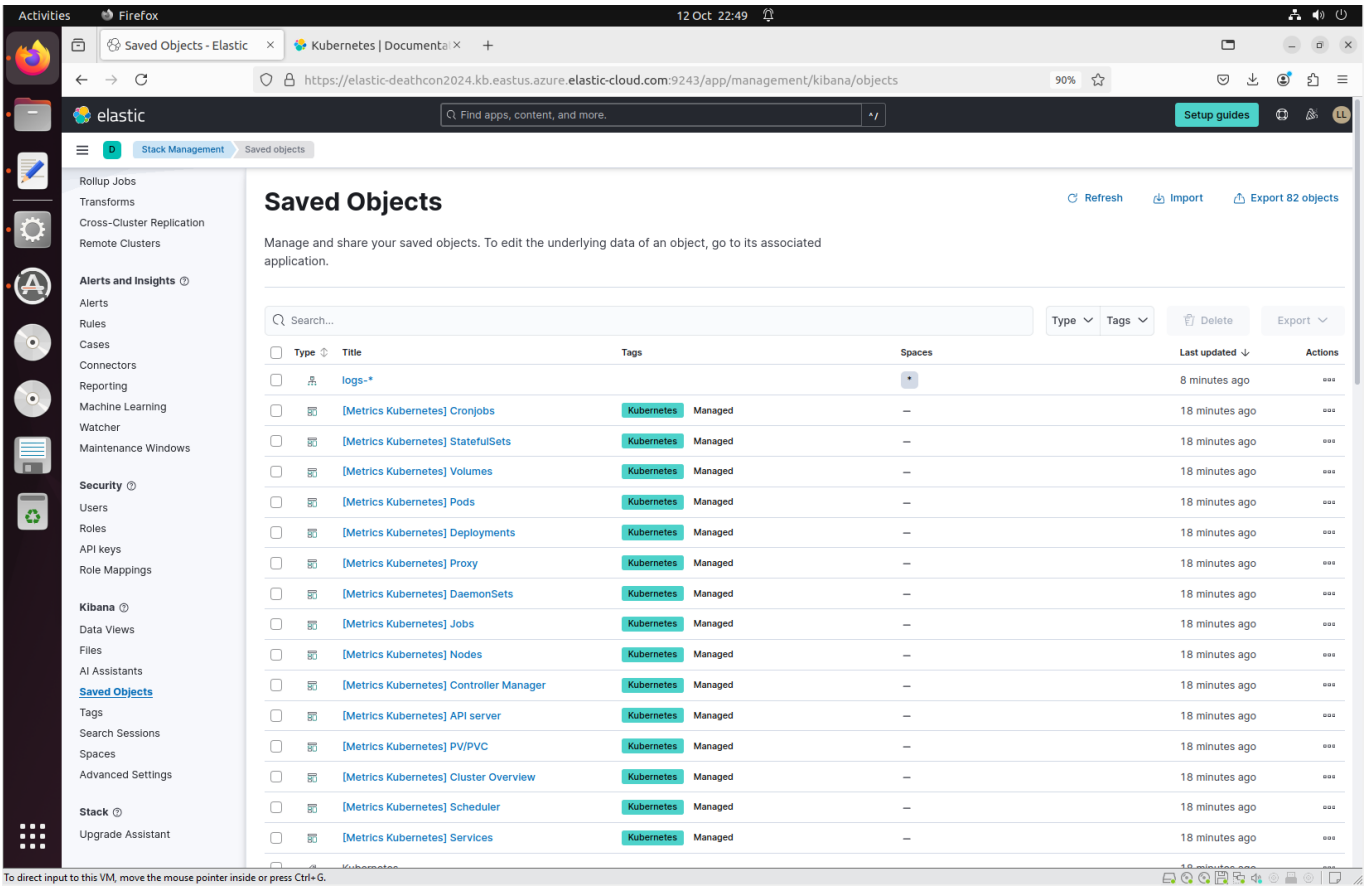


2. (Optional) Import the K8S Audit Logs Data View

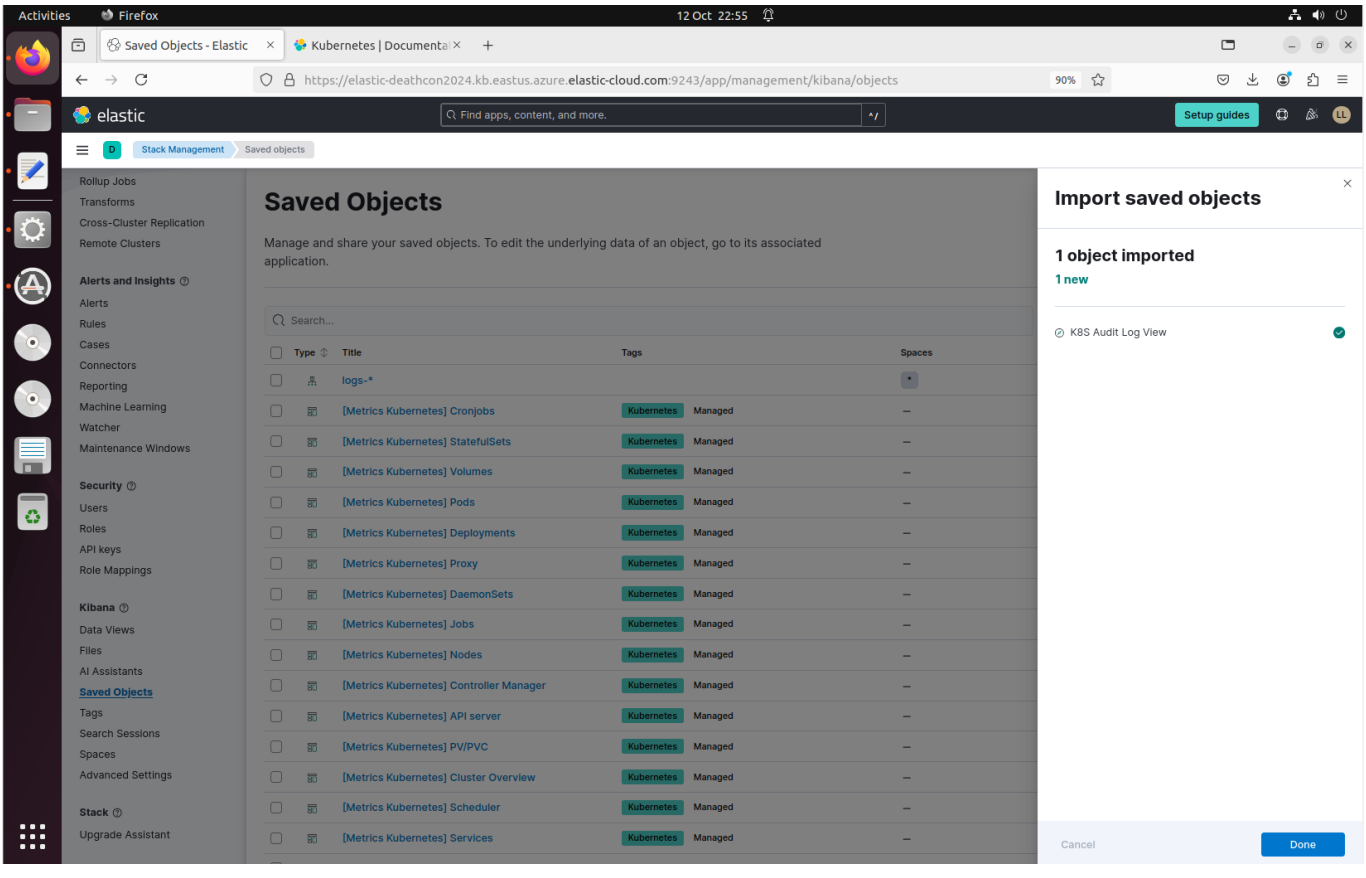
As an optional “Quality of Life” enhancement, feel free to import the data view provided as part in the student materials, which is optimised for Kubernetes Audit Log querying.

From the sandwich menu, navigate to Management > Stack Management.

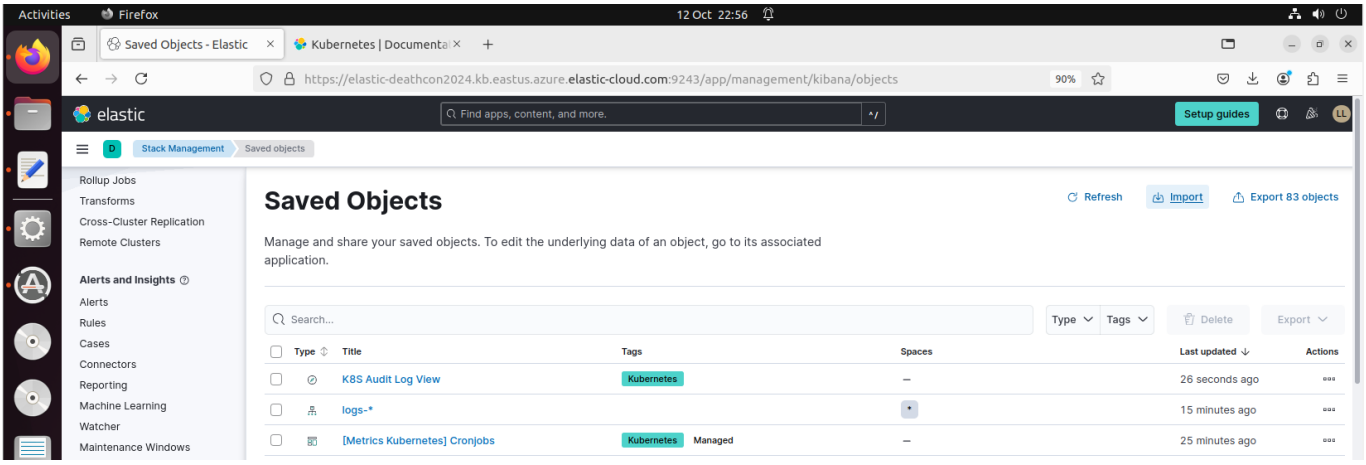
Under the Kibana section, select Saved Objects



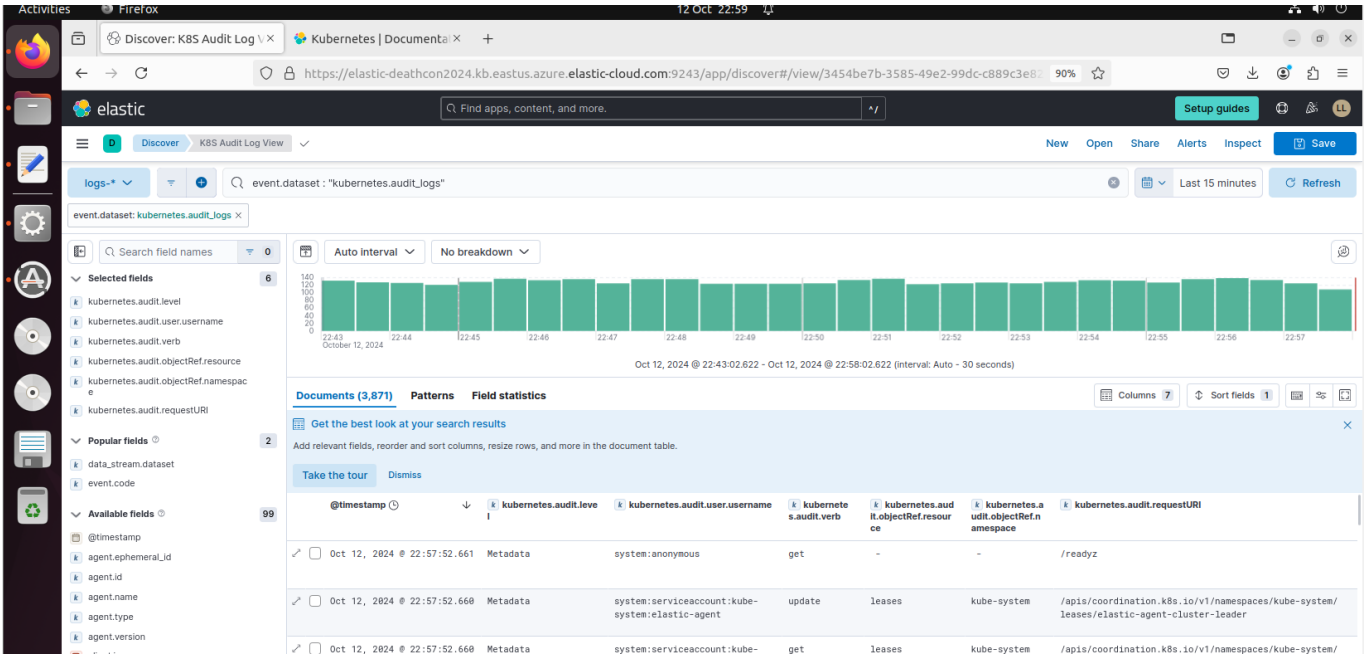
From there, click “Import” on the top right, choose “Request action on conflict” from the prompt, and select the `elastic-search-obect-auditlogsview.ndjson` file from your student materials.



Once this is loaded, hit “Done”, and you’ll have a new Object saved “K8S Audit Log View”



Clicking it, brings us to a more focused view, with only the relevant data filtered, and their fields of relevance set as columns:

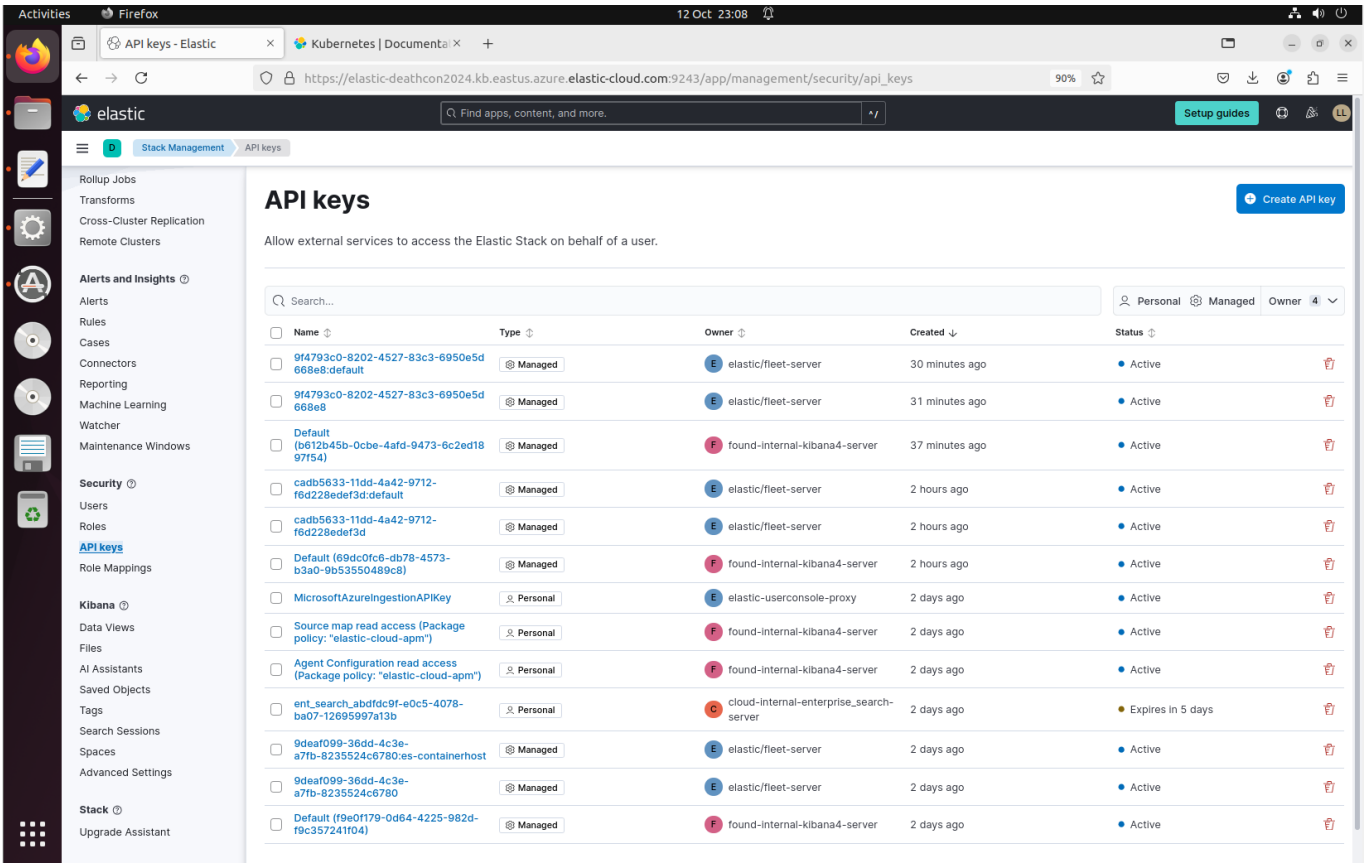


3. Connect Falco with the lab’s Elastic instance:

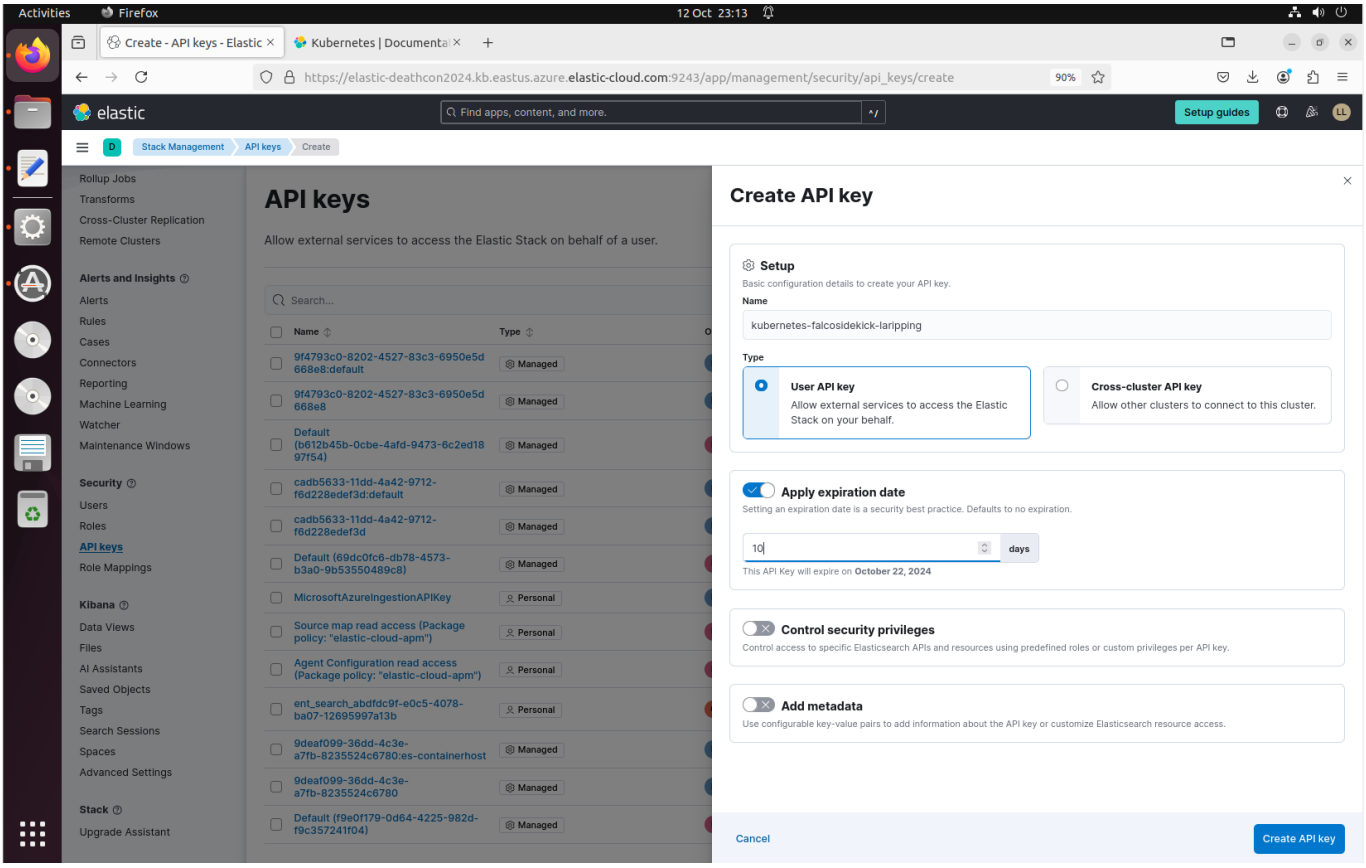
Now, we should also forward alerts raised by Falco into our deployment, to have all data flowing in the same place. This will require creating an Elastic API Key for Falcosidekick to use.

From the sandwich menu, navigate to Management > Stack Management.

Under “Security” select “API Keys”



Click the “Create API Key” from the top right, and provide a name with your own studentID such as the below one, leaving the Type set to “User API key”. Optionally, set an expiration date, to automatically stop clogging the shared Elastic instance with your Falco alerts, after you’re done with the lab:

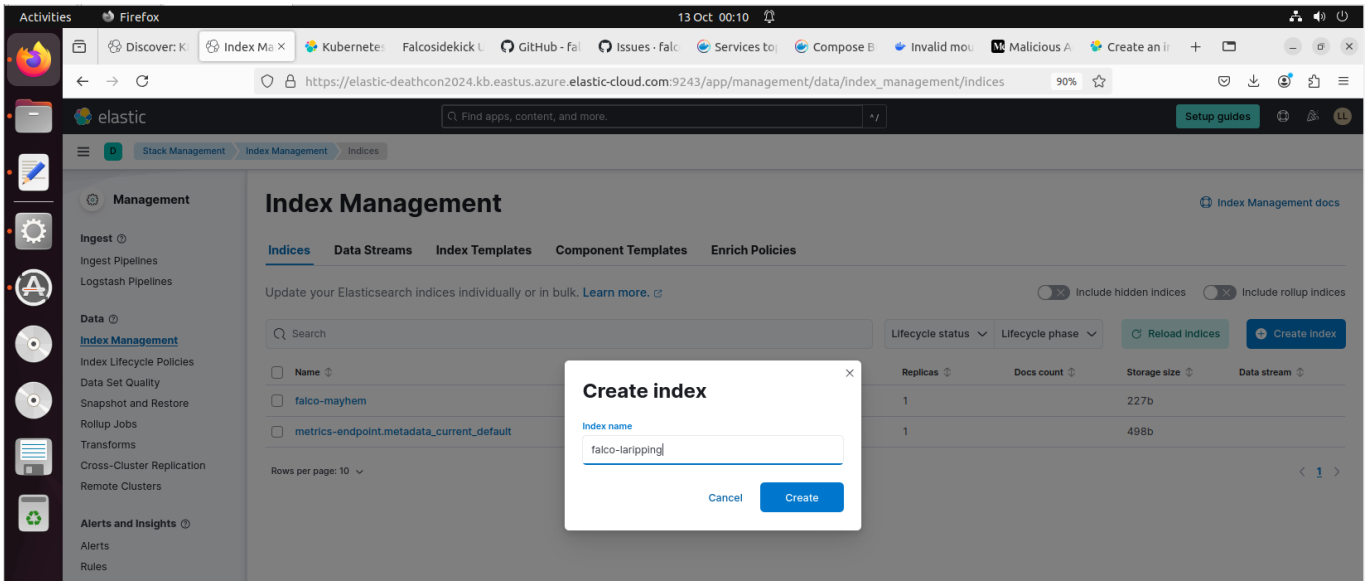


Hit “Create API Key” and copy the provided value somewhere temporarily.

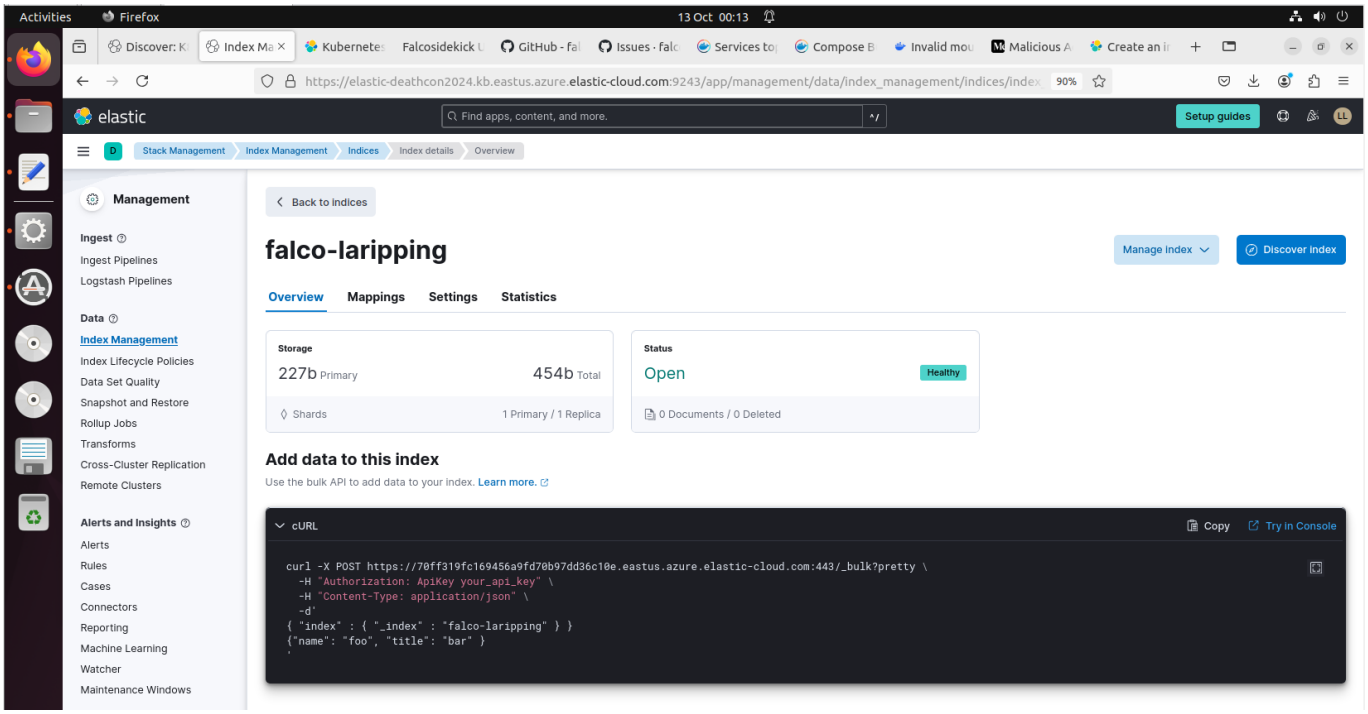
We also need to create an index in our Kibana, where the falco data will go into.

Depending on your preferred “difficulty level” for this workshop, you can choose to either separate your logs from those of other students, or merge them all into the same index for a more realistic SIEM experience.

- easy mode – to create our own falco index, let’s navigate to Management > Stack Management and under Data, let’s select “Index Management”. There, we click on “Create New Index” and provide a name in the format “falco-studentID”



Hitting “Create” and selecting the new index, gives us the endpoint where our falcosidekick should be pointed to. Let’s copy that value too.



- hard mode – to ship our falco alerts into the same stream as every other student, let's navigate to Management > Stack Management and under Data, let's select "Index Management". There, we expand the "falco-mayhem" index and copy the endpoint we see somewhere temporarily:

  https://70ff319fc169456a9fd70b97dd36c10e.eastus.azure.elastic-cloud.com:443

Now, let's revisit our falco setup. We need to tell falcosidekick to send data not only to the falcosidekick UI, but also to an Elastic instance. We read in the documentation that to achieved this, we need to provide a config file to falcosidekick, with the necessary pointers and authentication material…

Edit your `student-materials/falcosidekick-config.yaml` file, setting the fields under the `elasticsearch` key:

- `hostport:` the endpoint you copied above
- `index:` whether `falco-mayhem` or `falco-<studentID>` depending on your difficulty level
- `Authorization:` add the `ApiKey` you copied previously

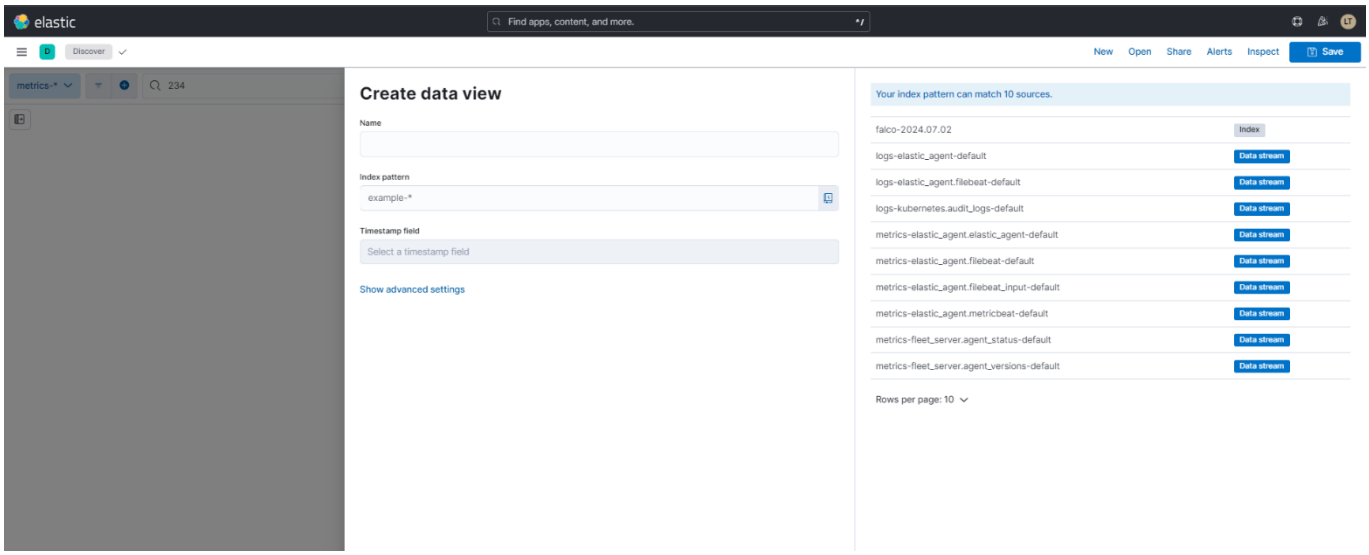
Now we also need to modify our docker compose setup to pass this new configuration to falco. Add the bolded lines below to `docker-compose.yml`

```
version: '3.3'
services:
    ...
    falcosidekick:
        container_name: falco-sidekick
        depends_on:
          redis:
            condition: service_started
        ports:
            - '2801:2801'
        image: falcosecurity/falcosidekick
        environment:
            - WEBUI_URL=http://falco-ui:2802

        entrypoint: ./falcosidekick -c /falcosidekick-config.yaml
        volumes:
            - ./falcosidekick-config.yaml:/falcosidekick-config.yaml

    falco-ui:
        ...
```

Restart the services. If you omit the --detach flag you can validate that falcosidekick is sending data successfully to Elastic, by seeing the 201 HTTP status codes upon new event generation.

```
$ docker compose down
$ docker compose up
...
[+] Running 4/4
 ✓ Network ubuntu_default    Created
0.1s
 ✓ Container redis           Created
0.1s
 ✓ Container falco-sidekick  Created
0.0s
 ✓ Container falco-ui        Created
0.0s
...
falco-ui        | 2024/10/12 23:16:55  NEW event 'event:f935d60a-1d19-466f-bfab-cce4cba8ccc9'
falco-sidekick  | 2024/10/12 23:16:55 [INFO]  : WebUI - POST OK (200)
falco-sidekick  | 2024/10/12 23:16:56 [INFO]  : Elasticsearch - POST OK (201)
```

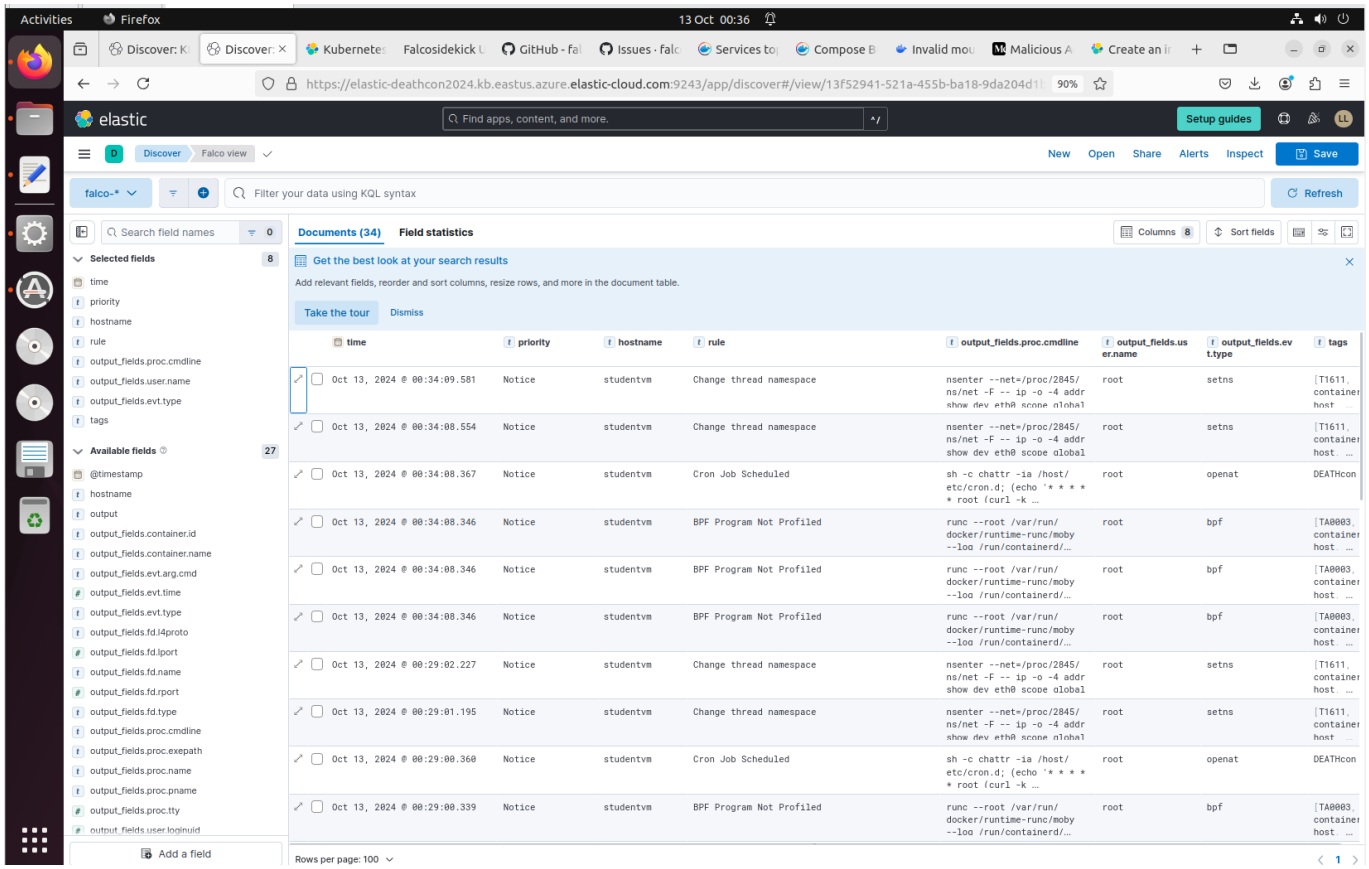
To view these events, we can create a new Data View on our Kibana. On the right hand side, the available log sources should include the `falco-mayhem` and your `falco-studentID` index - depending on your previous choice.



Hit "Save data view to Kibana"

4. (Optional) Import the Falco View

As in the previous part, we can import the `student-materials/elastic-search-object-falcoview.ndjson` Kibana View for an optimised Falco querying experience.



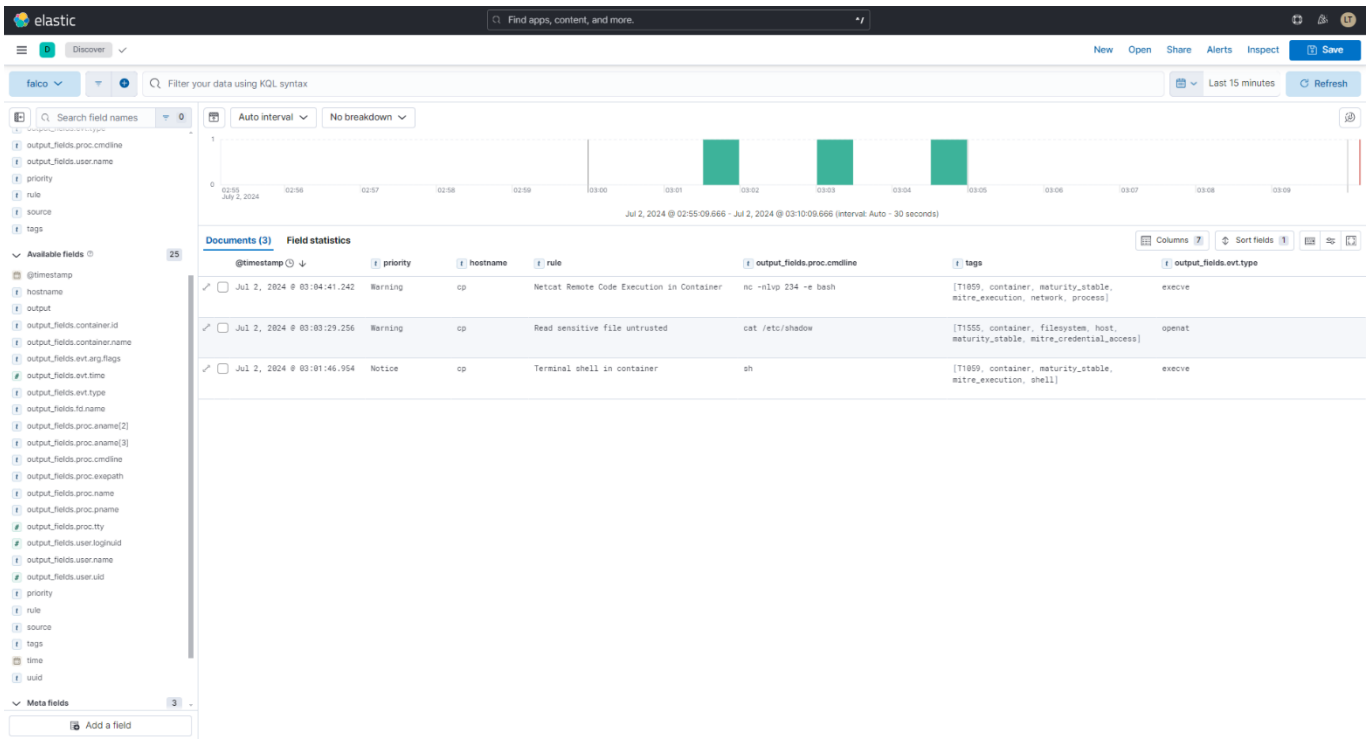
5. Confirm ingestion of Falco alerts into Elastic

The above steps already confirm ingestion of some Falco data into the Elastic deployment, but to be absolutely certain, let's perform some container-level attacks, and hunt for the logs in Elastic.

Shell into our netutils pod, and run a couple of typical OS level attacks:

```
$ kl exec netutils -ti – bash
root@netutils# cat /etc/shadow
root@netutils# nc -nlvp 123 -e bash
```

You should see these alerts forwarded into the new Data View within Elastic



1.2 Simulating Kubernetes Attacks with Leonidas

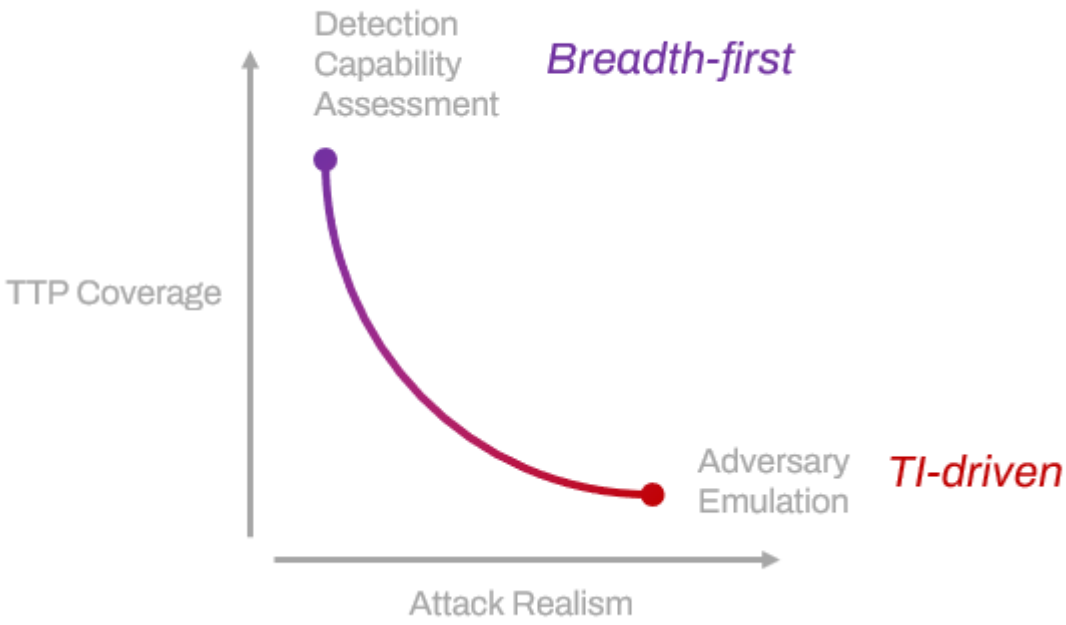
Having successfully set up basic security monitoring for our Kubernetes cluster, we can now focus our research into offensive operations. Reading up on attacks, understanding them, and re-producing them – with as much realism as safely possible - is crucial for engineering effective detections.

To simplify this attack execution phase, WithSecure has developed **Leonidas for Kubernetes**, an extension of the Leonidas cloud attack simulation framework allowing execution of Kubernetes attacks.

When offensive expertise is added into the mix for the purposes of collaborative attack simulations, these Purple Team exercises can broadly take two forms, depending on the objective at hand.

a) They could aim to cover as many TTPs as possible, with the purposes of baselining an organisation's detective capability. Executing each test case sequentially, in isolation

b) They could aim to realistically imitate the steps of a real-world threat actor, as documented by relevant threat intelligence reports



In this chapter we'll deploy, configure and use Leonidas, to demonstrate its capabilities. Afterwards, in chapter 1.3, we will explore how Leonidas can also be used for threat intelligence-driven adversarial emulations.

Deploy Leonidas

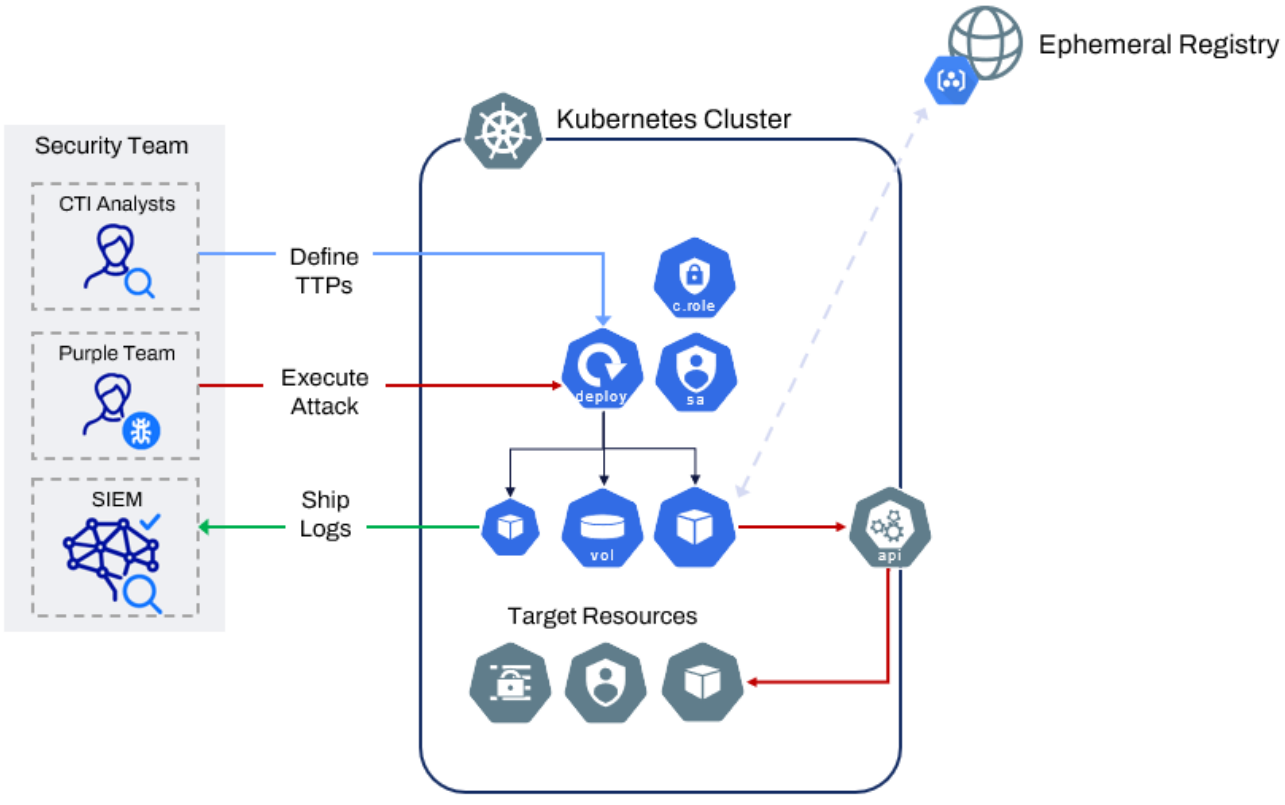
1. Clone Leonidas for Kubernetes

Just a Question of Time

At the time of writing this guide, efforts are underway to complete the push to the upstream repository, which will allow a simple git clone https://github.com/WithSecureLabs/Leonidas. However, to address the scenario that this is not yet completed by the time this guide gets to you, a version of Leonidas including the Kubernetes features can be found in your student materials: `leonidas-kubernetes-master.tar.gz`

Without looking too deep under the hood, the project's README gives us a high level idea of the architecture for Leonidas' Kubernetes implementation. They key points to notice are that:

- Attack test cases ("definitions") are written in YAML, in a simple format
- Leonidas can then generate Python code for these test cases, which extends a Flask application
- The complete Leonidas Python application is **containerised,** in an ephemeral image that can be pushed to any registry
- As such, it can be deployed **inside the target cluster**, as Kubernetes resources (mainly, a Deployment)
- We can interact with the Leonidas REST API over a Service, accessible from outside the cluster using a simple `kubectl port-forward` command



2. Review a couple of YAML definitions

To better understand Leonidas' capabilities, let's review the YAML files for two of the attack definitions included out of the box, under the `definitions/` directory:

- `discovery/enumerate-nodes.yml`
    - After the metadata such as the author, and description, the first point to focus on is `code` field, under `executors.sh` . In this case, it's a simple and intuitive `"kubectl get nodes"` command – which will be performed from within Leonidas's pod
    - To succeed with this operation we would need some permissions: Notice, under the `permissions:` block, a single entry consisting of a verb & a resource, defined in an API group. In this case, the permission required is cluster-wide (`namespaced: false`), and as such the test case will require Leonidas to be deployed with cluster-wide scope, as we'll see later.
    - In the `detection:` field, we notice how the verb to match is `list`, contrary to the `get` action specified by the kubectl command… This is just one of the many caveats of Kubernetes attack detection, that can only be discovered through practical experimentation!
- `privilege-escalation/privileged-container.yml`
    - This test case is a bit more complicated, as it creates a specially-crafted resource inside the cluster – a container with the `privileged:true` field in its `spec:`.
    - As such, it relies on a single argument of type `file` whose default value indicates the entire contents of the `custom.yml` that will be passed to the `kubectl apply -f` command
    - This time, the RBAC `permissions:` to successfully complete the operation and create the container are two – both of them scoped to a namespace only, and not requiring cluster-wide deployment
    - Finally, note the special `privileged:true` condition in the detection. Besides the verb & resource, our contributions to the Sigma backend pipelines allow matching of certain fields within the JSON documents of audit logs ingested in Elastic – more on that later.

3. Familiarise with the generator utility

It's time now to get more hands-on. Leonidas comes with a `generator.py` utility for various generation steps. Let's install this and familiarise with it. First, we install `poetry`:

```
$ curl -sSL https://install.python-poetry.org | python3 -
...
To get started you need Poetry's bin directory (/home/ubuntu/.local/bin) in your `PATH`
environment variable.

Add `export PATH="/home/ubuntu/.local/bin:$PATH"` to your shell configuration file.

Alternatively, you can call Poetry explicitly with `/home/ubuntu/.local/bin/poetry`.

You can test that everything is set up by executing:
`poetry --version`
```

Once we update our `.bashrc` as advised, we can proceed to invoke poetry and launch the generator utility

```
$ cd generator
$ poetry install
$ poetry run ./generator.py
Usage: generator.py [OPTIONS] COMMAND [ARGS]...

Options:
  --config PATH                   Path to config file  [default: config.yml]
  --install-completion [bash|zsh|fish|powershell|pwsh]
                                  Install completion for the specified shell.
  --show-completion [bash|zsh|fish|powershell|pwsh]
                                  Show completion for the specified shell, to
                                  copy it or customize the installation.

  --help                          Show this message and exit.

Commands:
  build-image        Build and push the Leonidas container image.
  definitions        Pretty-print definitions dictionary
  docs               Generate the leonidas documentation...
  generate-aws-api   Generate the AWS Leonidas API
  generate-kube-api  Generate the Kubernetes Leonidas API
  iam-policy         Print the AWS IAM policy necessary to execute all the...
  kube-resources     Print the Kubernetes YAML resources
  leo                Generate the configuration file for Leo, the CLI tool...
  rbac-policy        Print the Kubernetes RBAC policy as YAML, including
                     all...

  serverless-config  Print the Serverless framework configuration used to...
  sigma              Generate the Sigma rule definitions
  validate           Validate the test definitions
```

4. Generate Definition Documentation

To test-drive generator, let's go ahead and create browsable documentation for our attack definitions. As per the Leonidas README:

```
poetry run ./generator.py docs
```

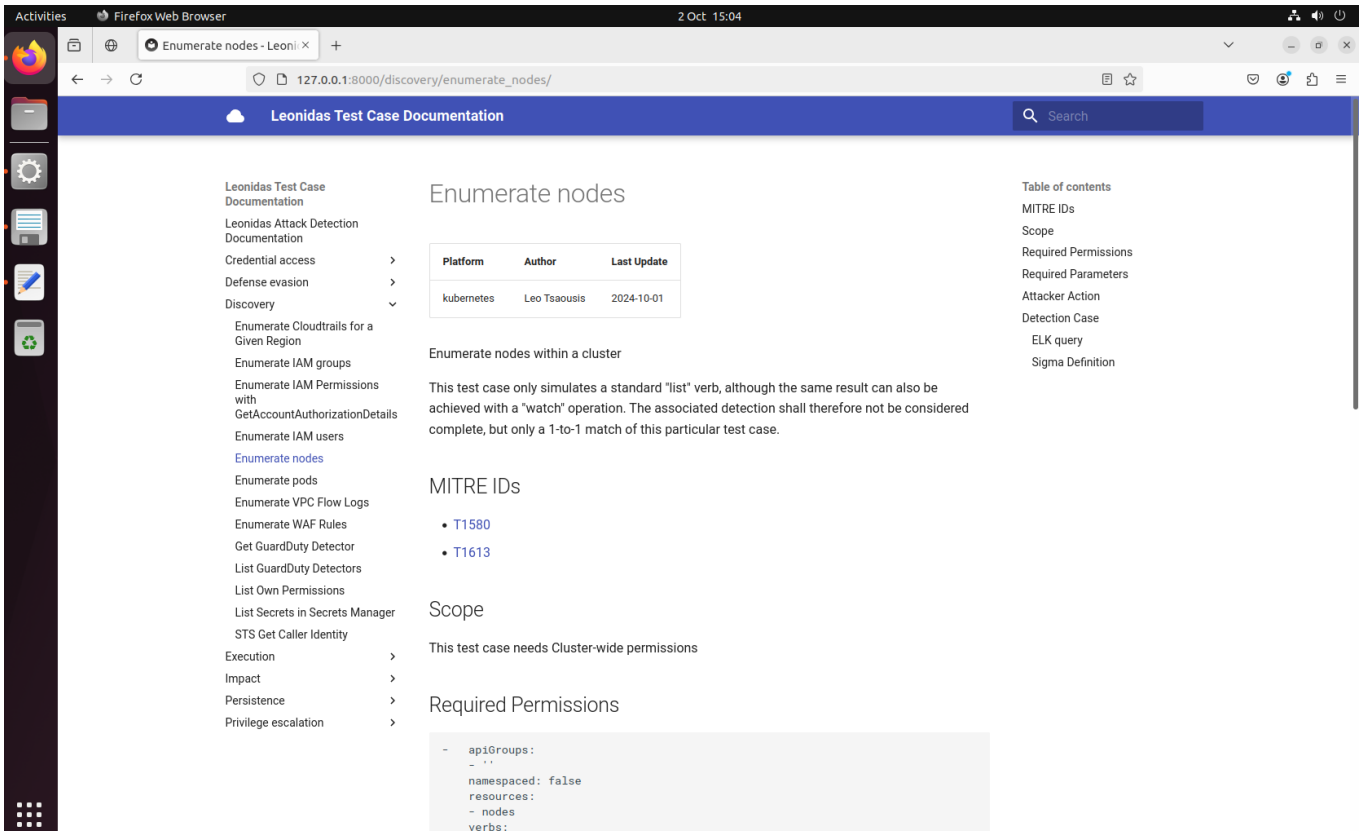
This will produce markdown versions of the documentation available at `output/docs`. This can be uploaded to an existing markdown-based documentation system, or the mkdocs command can be used to create a prettified HTML version of the docs. `mkdocs` has been installed via `pip`, as part of generator's dependencies, so we can run it using poetry, from the generator's directory

```
$ poetry run bash -c 'cd ../output && mkdocs build'
```

This will create an `output/site` folder containing the HTML site. It is also possible to view this locally by running the following command in the same folder:

```
$ poetry run bash -c 'cd ../output && mkdocs serve'
```

This should spin up the mini page in port 8000, where we can browse documentation for attack definitions, including the ones we just examined:



Let's now go ahead and run these attacks, within our cluster.

5. Deploy Leonidas into cluster

To execute these test cases, we need to deploy Leonidas into our cluster, and interact with it. Following the instructions in the `deploying-leonidas.md` documentation page, let's proceed to configure and run the generator utility, in order to build the Python code from the YAML definitions as well as the Kubernetes resource manifests required.

Let's start by modifying `config.yml` to specify our ephemeral image repository, and the namespace which Leonidas should be deployed into:

```
$ cat config.yml
...
# Kubernetes namespace to generate role into (if not specified the "default" namespace is used)
namespace: persepolis
# Image registry to push & pull the image
image_url: ttl.sh/leonidas-<studentID>:15m
```

Remember: Ephemeral ≠ Private!

To keep things simple and support ephemeral images, Leonidas & generator are built to work with a **public** repository such as ttl.sh. Setting a short lifespan, as recommended, such as 15 minutes or a few hours, would normally reduce the exposure of your own image to third-party requestors, (and most importantly, your exposure to third-party images!). For a workshop scenario like this one, however, it is very likely that Leonidas images pushed by participants might collide. In the worst case, *this could result in arbitrary code Python executed inside your cluster!* To avoid such cases, you are encouraged to append a unique identifier on your images (indicated by the `<studentID>`) in order to ensure your image does not collide with other students' ones. The generator utility will take this value into account when creating the Kubernetes manifests, meaning you only need to define this once, in `config.yaml`.

Next step is to generate the Python code from our YAML definitions:

```
$ poetry run ./generator.py generate-kube-api
Generating API locally
API generation complete - 18 cases generated
```

This will create one Python file for each Tactic, under the `output/leonidas/api/` directory

```
$ ll ../output/leonidas/api/
total 88
drwxrwxr-x 2 ubuntu ubuntu  4096 Oct  2 15:38 ./
drwxrwxr-x 3 ubuntu ubuntu  4096 Oct  2 15:38 ../
-rw-rw-r-- 1 ubuntu ubuntu  1797 Oct  1 19:19 api_base.py
-rw-rw-r-- 1 ubuntu ubuntu 11700 Oct  2 15:38 credential_access.py
-rw-rw-r-- 1 ubuntu ubuntu  5591 Oct  2 15:38 defense_evasion.py
-rw-rw-r-- 1 ubuntu ubuntu  7914 Oct  2 15:38 discovery.py
-rw-rw-r-- 1 ubuntu ubuntu  9583 Oct  2 15:38 execution.py
-rw-rw-r-- 1 ubuntu ubuntu  8303 Oct  2 15:38 impact.py
-rw-rw-r-- 1 ubuntu ubuntu     0 Oct  1 19:19 __init__.py
-rw-rw-r-- 1 ubuntu ubuntu  3002 Oct  2 15:38 persistence.py
-rw-rw-r-- 1 ubuntu ubuntu  7043 Oct  2 15:38 privilege_escalation.py
-rw-rw-r-- 1 ubuntu ubuntu  8575 Oct  1 19:19 utils.py
```

The Python application is ready. Let's containerise it, by building the Leonidas Image using the VM's Docker engine, and push it to ttl.sh – all in one go:

```
$ poetry run ./generator.py build-image
```

Troubleshooting Tip

If you get a `BuildError` from docker, with error messages relating to networking, then restart the docker engine and try the above command again. Don't forget to restart minikube!

Once this eventually finishes, we can confirm our new image locally

```
$ docker image ls
REPOSITORY                      TAG       IMAGE ID       CREATED          SIZE
ttl.sh/leonidas                 15m       5b1a9a12f62d   3 minutes ago    1.14GB
...
```

With the image uploaded, we can now author the manifests for Leonidas' Kubernetes resources. As we want to deploy the Leonidas pod with cluster-wide permissions first, we can either explicitly set the `--clusterrole` flag in the below command, or skip all flags, defaulting to cluster-wide operation

```
$ poetry run ./generator.py kube-resources > ~/leonidas-manifests.yml
```

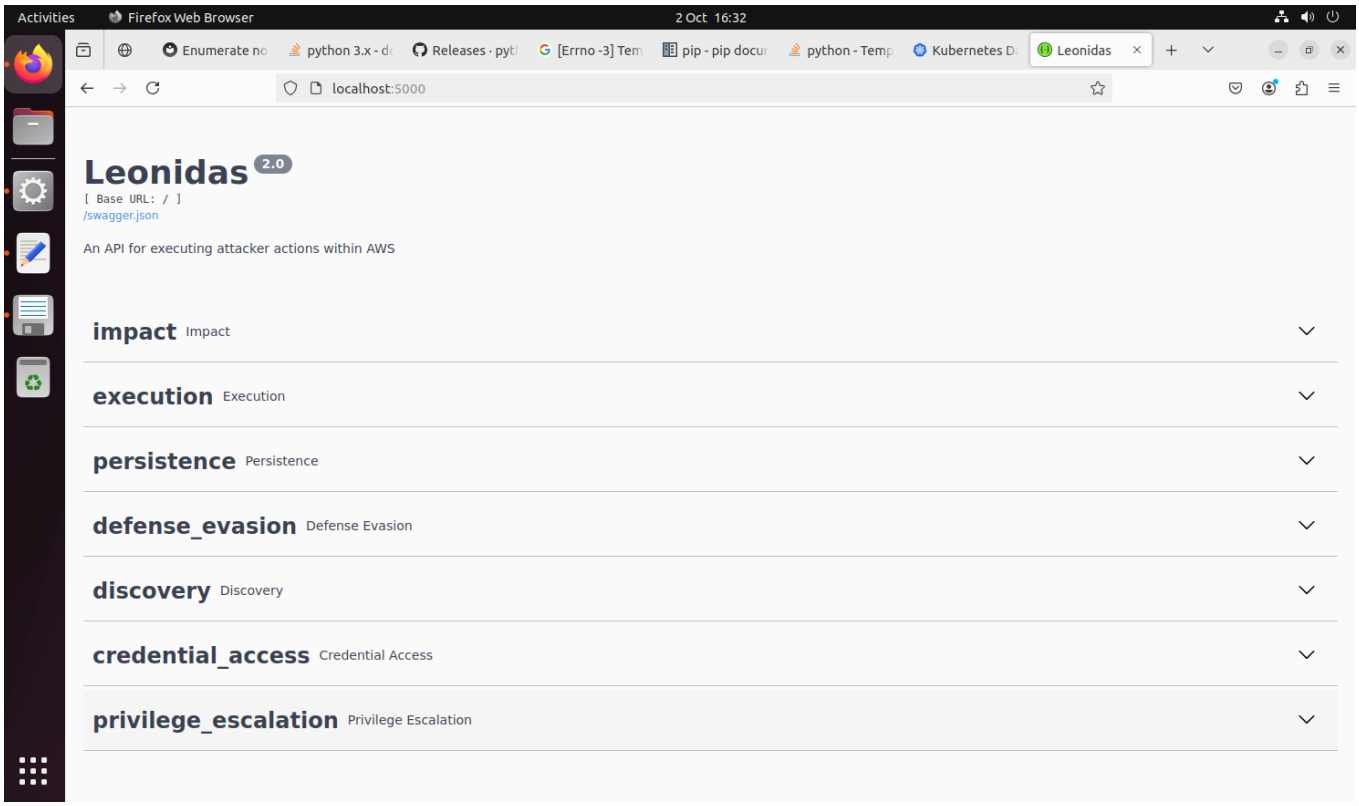
Finally, we'll create these resources into our cluster

```
$ kubectl -f ~/leonidas-manifests.yml apply
```

Once all components have been started, we'll expose the Leonidas' Flask web service locally, to interact with it through the web browser:

```
$ kubectl port-forward deployment/leonidas-deployment 5000:5000 -n persepolis
```

The familiar Flask interface of Leonidas should greet us:

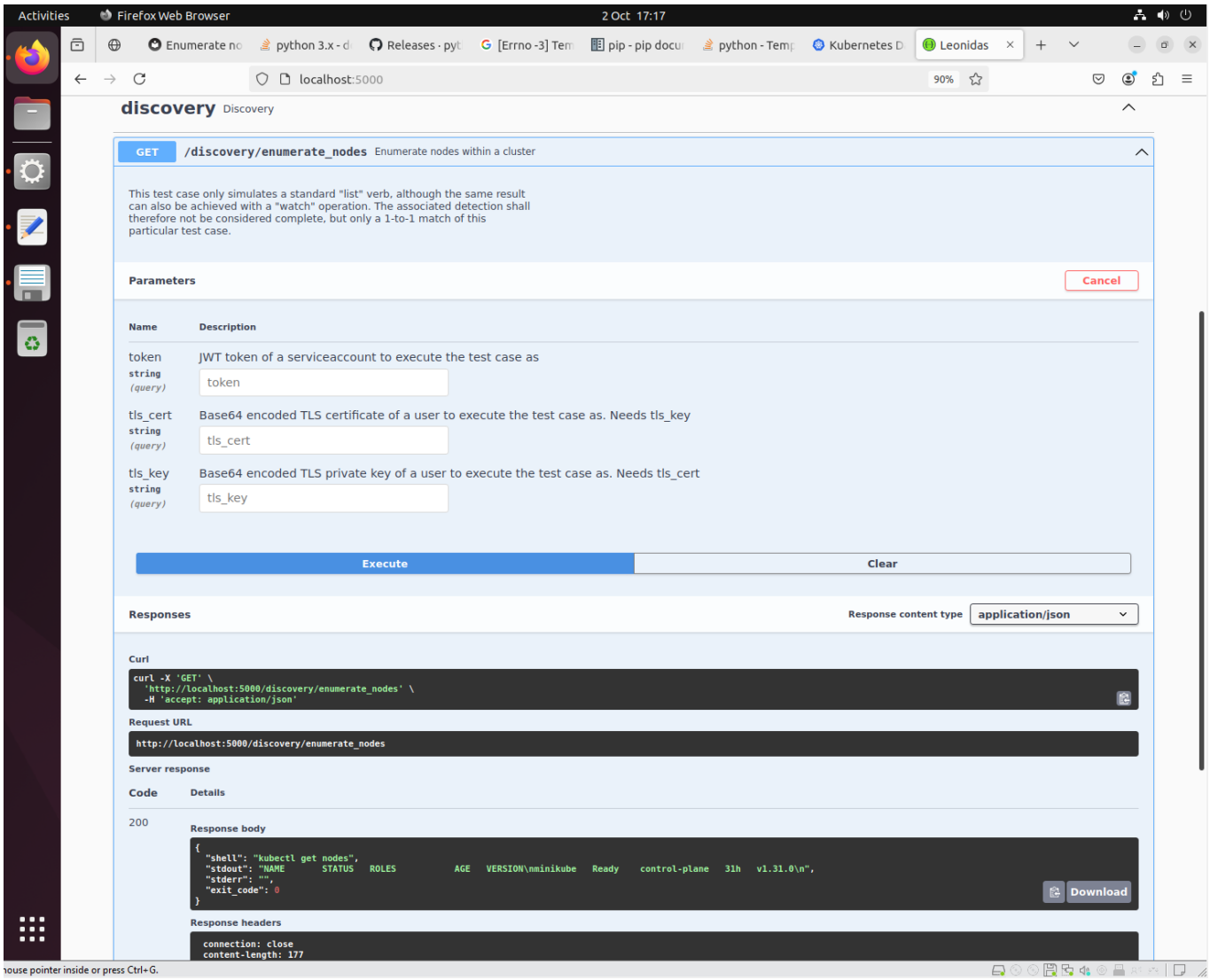


This web UI is one of the clients we can use to interact with Leonidas. Let's go ahead and execute our test cases in question, through the UI.

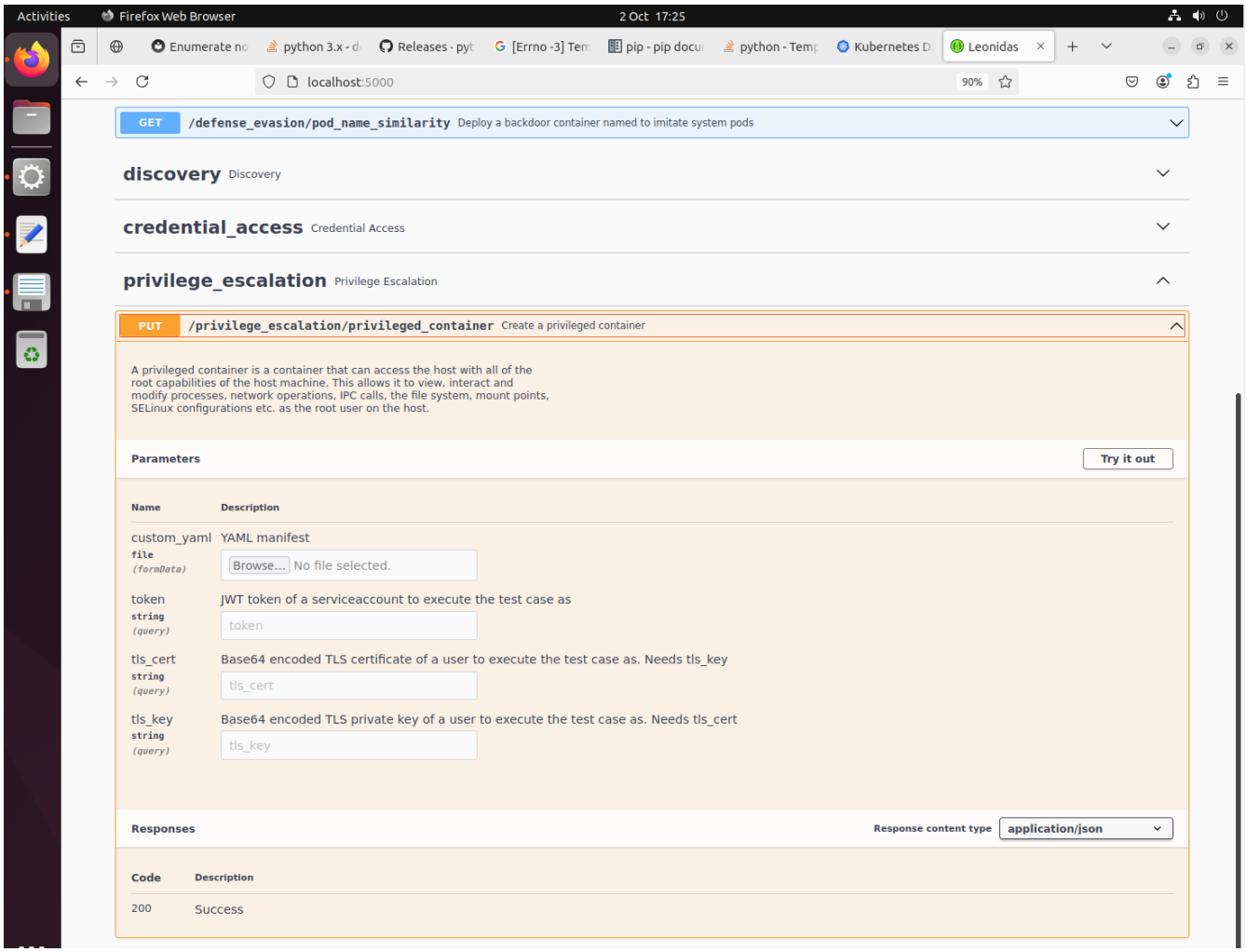
Execute Test Cases

1. Execute Test Cases via the Web UI

First, we locate the `enumerate_nodes` test case under Discovery, and proceed to hit "Try it out", then "Execute" without supplying any values for the parameters. These are the standard parameters added to every test case for the purposes of identity assumption, as we'll see in the next step. As they are optional, the test case is implemented as a GET request (blue colour)

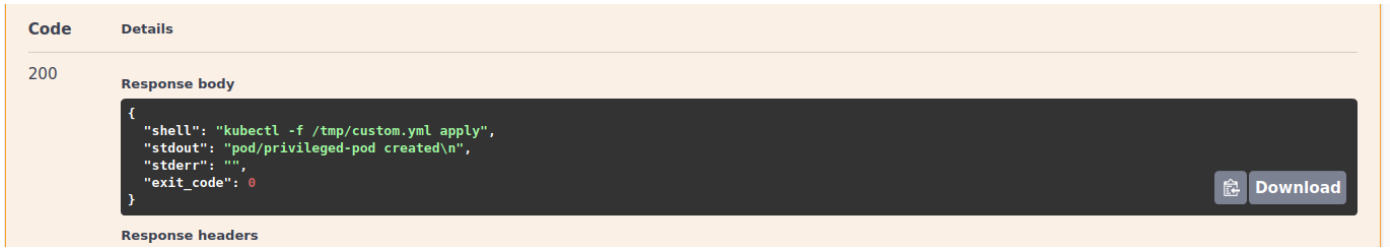


As for the `privileged_container` test case, it defines an argument other than the identity assumption parameters, and since this argument's value is of type "file", the test case is implemented as a PUT request (denoted by the orange colour)



While the default value configured would do the trick, let's supply a slightly modified version of that file to test out user-provided argument values:

Hitting the "Browse" button to choose the `my-privileged-pod.yaml` file instead, and proceeding to "Execute" leads to the customised privileged pod be deployed, as expected:



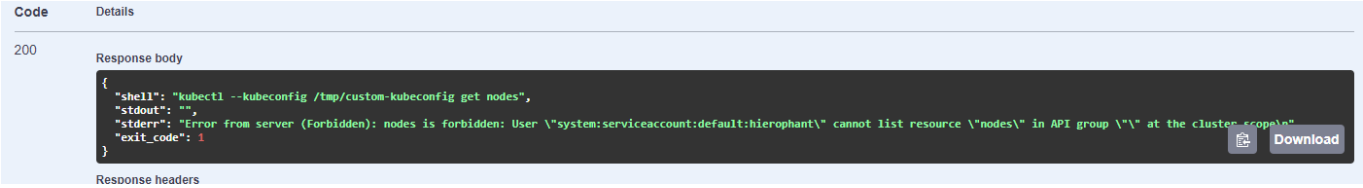
2. Execute Test Cases via the Web UI, assuming another Identity

As described in the `using-leonidas.md` document of the project, Leonidas can also assume another identity to launch attacks, different to its Service Account. By supplying the appropriate credentials – a JWT token to for another service account, or a Base64-encoded X509 certificate and key – the identity of a different entity can be used.

To see this in action, first generate a token for a new Service Account

```
$ kubectl create sa hierophant
serviceaccount/hierophant created

$ kl create token hierophant --duration=9999h
eyJhbGciOiJSUzI...
```

Then repeat the test case passing these alternative credentials.



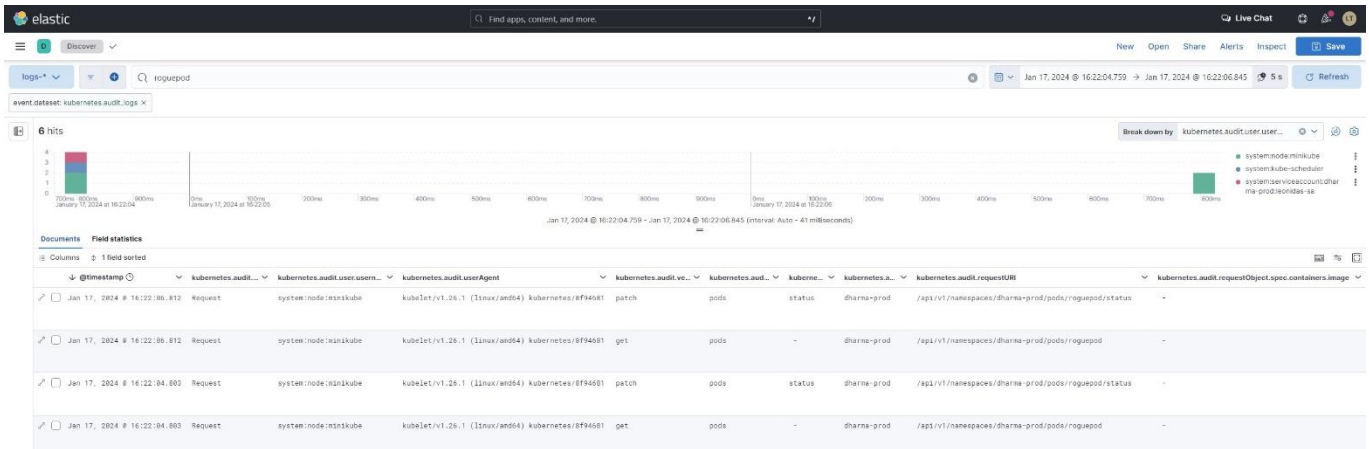
Expectedly, the attack has now failed, given the alternative Service Account does not have the cluster-wide permission to list nodes

Detection Engineering

With the ability to simulate attacks against a test environment, and the collection of appropriate logs for most of the attack surface of interest, we now have enough in our hands to engineer detections for Kubernetes attack TTPs.

1. In Elastic, see their audit events of the test cases we executed

Back in our “K8S Audit Logs” View, a simple query in the searchbar, for keywords such as argument values we used previously, should give us the JSON documents of the respective events



This signature left in the logs by each attack test case, can be captured by a Sigma rule, for easier distribution and management. And with the necessary additions to the SigmaHQ/pySigma-backend-elasticsearch project now merged, we can leverage the Kubernetes:Audit log source, to author these rules:

```
...
logsource:
    category: application
    product: kubernetes
    service: audit
detection:
    selection:
        verb: 'delete'
        objectRef.resource: 'deployments'
    condition: selection
...
```

Vendors, Products, Solutions…

As with every other attack surface, Kubernetes is not short of security monitoring technologies, each shipping its own set of pre-built detections.

Being an open source tool, one of Leonidas' principles was to be Vendor Agnostic. As such, when it comes to detection rules, Leonidas opts for the Sigma format to describe them – an open, and product agnostic format for distributing alert logic, without complying to a specific vendor language or product's specification.

While the rest of this section will use Sigma rules and their integration with Elastic, some alternatives for popular monitoring technologies are provided below, along with pointers for their pre-made alerts to get you started for your own technology stack:

- **Elastic** – The Elastic Kubernetes Integration, which we've used for audit log ingestion, actually ships with some alerts on top of those audit logs
- **Falco** – Falco's default YAML rules includes a few detections alerting upon system events observed, such as network connections to the K8S API server
- **Tracee** – Similarly, Tracee also defines build-in detections for system events observed, such as network connections to the K8S API server
- **GuardDuty** – For a managed cluster hosted on AWS, GuardDuty detections exist for suspicious Kubernetes activity, using EKS Audit Logs

The generator utility can also generate the Sigma rules for all attack definitions available, using the `detection:` block in each YAML. Let's give this a go:

2. Generate Sigma Rules

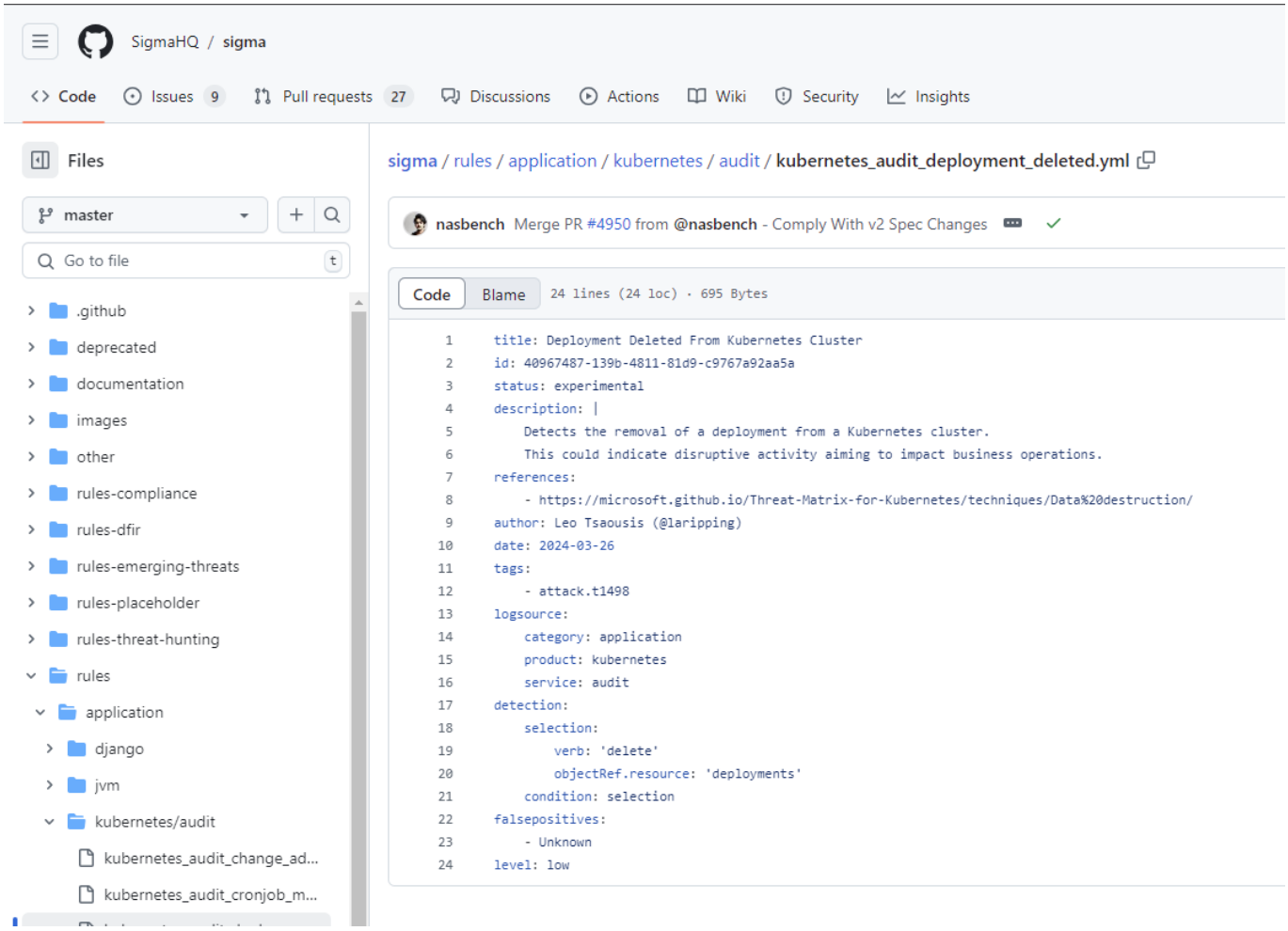
As per the instructions in Leonidas' README:

```
$ cd generator
$ poetry run ./generator.py sigma
Generating Sigma rules
Generated 68 sigma definitions
```

This generated the Sigma rules not just for Kubernetes, but for all the test cases currently shipped with Leonidas, including the 50 or so AWS ones.

All of them can be found in the `output/sigma` directory, organised per tactic.

If we examine a Kubernetes one specifically (e.g. `impact/delete_deployment.yml`), we can see it's in the official Sigma format, distributable and parseable by all downstream tools integrated with Sigma. As such, these rules have been merged upstream into the SigmaHQ/sigma project.



3. Load the Detections for our test cases in question, into Elastic

Having a technology-agnostic detection in our hands, we can now convert it to our particular SIEM of choice – which in our lab is Elastic - and implement the alert. To simplify this downstream conversion, the Sigma ecosystem offers "backends" that the CLI can understand, including an Elasticsearch one. Let's use the CLI to covert our rules to actual Lucene Queries:

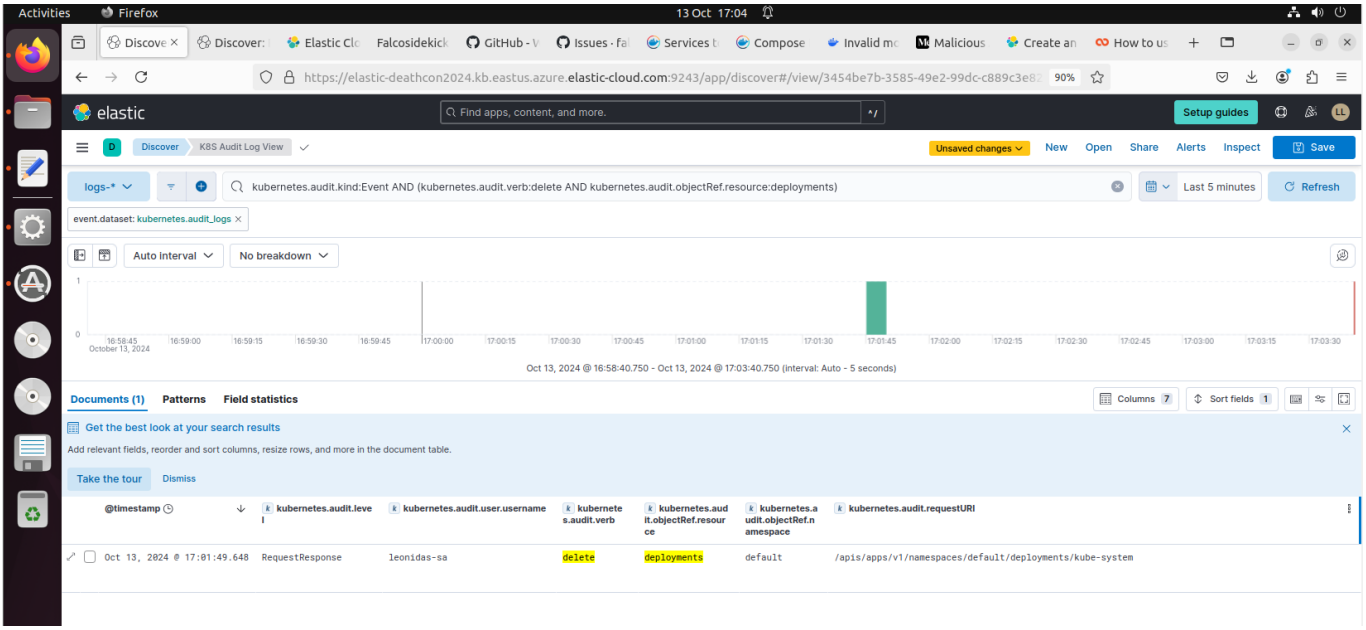
```
$ cd output/leonidas
$ poetry add sigma-cli
$ poetry run sigma plugin install elasticsearch

$ poetry run sigma convert -t lucene -p ecs_kubernetes ../sigma/impact/delete_deployment.yml
Parsing Sigma rules  [####################################]  100%
kubernetes.audit.kind:Event AND (kubernetes.audit.verb:delete AND
kubernetes.audit.objectRef.resource:deployments)
```

Note that alternatively, we could also use the online converter for this process, using the static YAML for our Pipeline:

Shortened link of sigmaconverter.io self-contained rule link : https://tinyurl.com/deathcon-sigma

4. Run the Query on ingested logs

To validate this detection, we can paste the above `sigma convert` output into the Discover search bar, and confirm some matching activity:



1.3 Emulating Real-World Adversaries

We now have the building blocks to create complex attack chains, including steps focusing on different attack surfaces. As Leonidas is an API, there are various clients we can use to interact with it over HTTP, whether manually or programmatically. Jupyter notebooks is one of those clients, which can also offer additional benefits to Blue/Purple teams: visual, reproducible attack narratives, stored and maintained as text. And these allow designing "Execution Runbooks" for our drills, or "Emulation Plans" reproducing threat actor campaigns.

Leonidas & Jupyter Notebooks

Let's first have a look at the notebooks available out of the box.

Within the `jupyter/threat-actors/` directory inside the Leonidas codebase, you'll find the OF815 Notebook – a runbook of a fictitious threat actor getting access to a Kubernetes cluster and running some attacks.

Let's see how that looks like in Jupyter

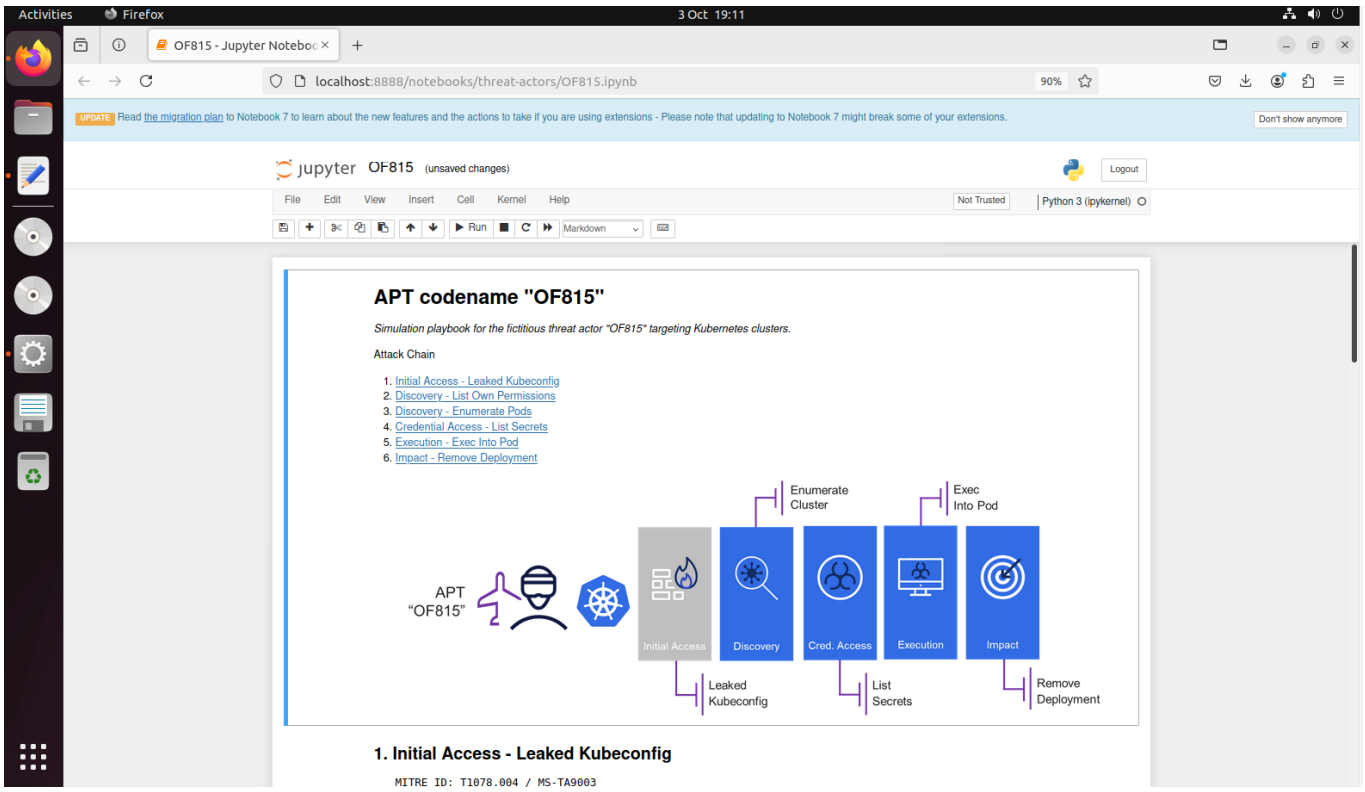
1. Start Jupyter Notebook

```
$ cd jupyter
$ poetry lock
$ poetry install
$ poetry run jupyter notebook
```

Troubleshooting Tip

If `poetry install` fails, run it a few more times

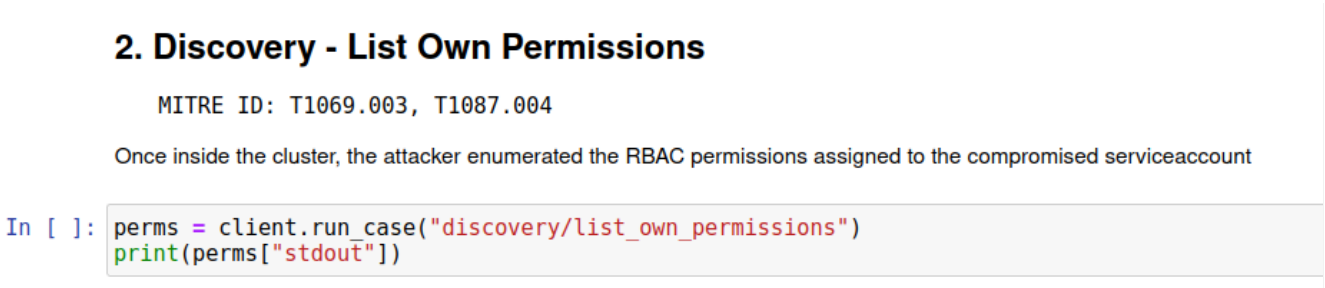
This should launch the notebook UI in the browser, where we can navigate to the `threat-actors/` directory and load `OF815.pynb`



While we could interact with Leonidas from the notebook directly, using an HTTP client such as `requests`, we have instead opted to "keep it clean" and abstract away the Request / Response parsing into a simple client defined in `jupyter/lib/kubeclient.lib`.

2. Examine the contents of the `KubeClientLib` class

This client can be instantiated early in the notebook, pointing it to the URL where Leonidas listens, enabling subsequent `run_case("tactic/test_case", args={})` calls that simplify interaction:

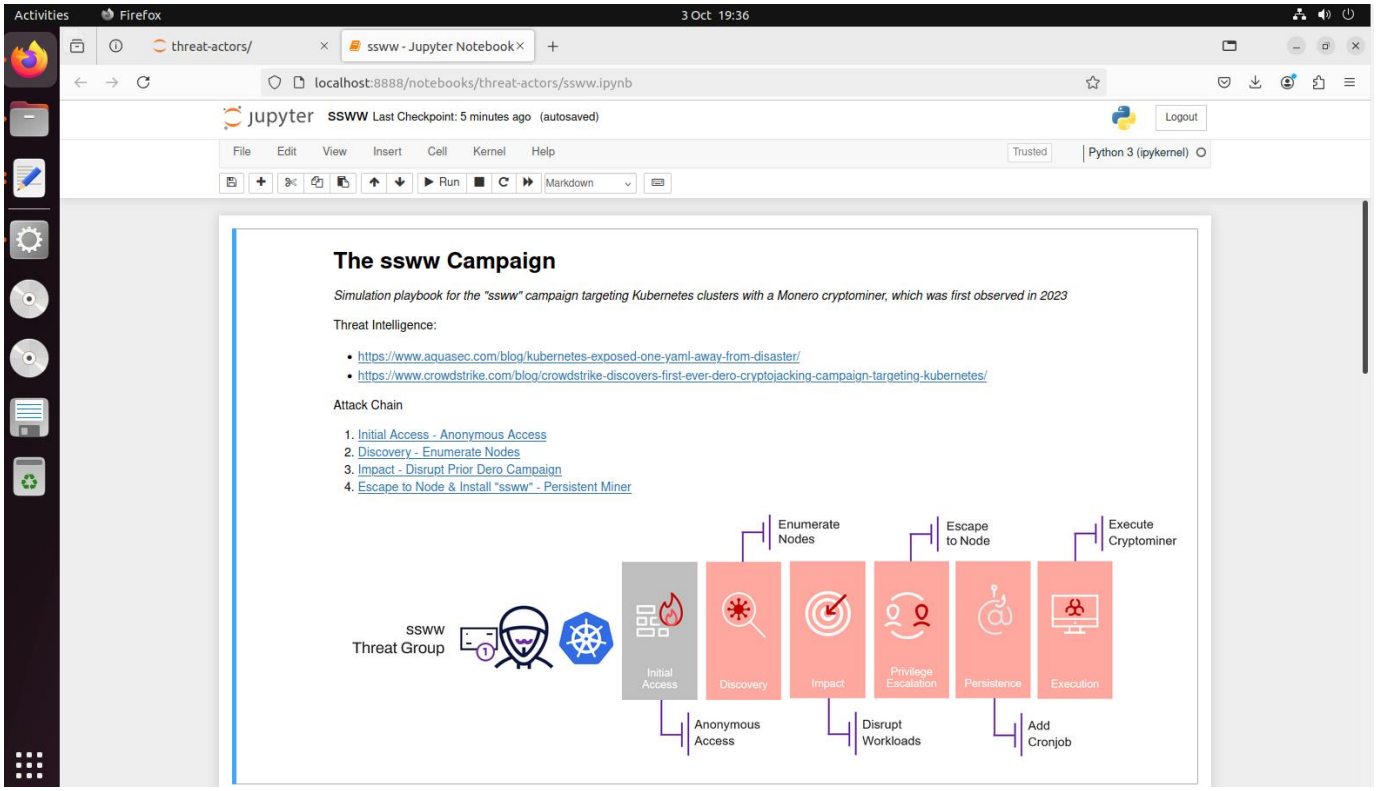


SSWW Run Trough

As part of the lab materials, you are also provided an exclusive, unreleased second playbook. This simulates a real-world adversary which was observed in 2021, dubbed SSWW

1. Merge the `jupyter` directory from the student materials, into the `leonidas/jupyter`
2. Load the new SSWW playbook under `threat-actors/` directory

The introductory header of the notebook will give you some background on the campaign and its operations



To simulate the TTPs of this threat actor, we have developed 3 more Leonidas attack definitions, which are also exclusive to this workshop and not published yet on the GitHub:

- Anonymous Request
- Delete DaemonSet
- Deploy the SSWW DaemonSet

To use them, we first need to write the YAMLs, convert to Python code, and build the new Leonidas image re-deploying into the cluster.

3. Merge the `definitions` directory from the student materials, into the `leonidas/definitions`
4. Re-deploy Leonidas with these new test cases

```
$ cd generator/
$ poetry run ./generator.py generate-kube-api
$ poetry run ./generator.py build-image
$ kubectl delete pod -n persepolis -l app=leonidas # deletion triggers fetching of new image
```

Once the new Leonidas pod is re-deployed, its status will be Running, with an age of a few seconds…

```
$ kubectl get pods -A
NAMESPACE        NAME                                    READY   STATUS    RESTARTS       AGE
...
persepolis       leonidas-deployment-5bbfff599d-h5mhs    2/2     Running   0              24s
```

…and the new `initial_access` API group will appear under in Flask UI:



5. Provide the appropriate RBAC permissions to Leonidas

In order to exercise some of our detections, we need some of the attacks to succeed.

As such, we'll go ahead and re-generate the RBAC policy for Leonidas, based on this new list of test cases defined. Once again, we'll operate with cluster-wide privileges, and not constrained within a certain namespace:

```
$ poetry run ./generator.py rbac-policy > ~/leonidas-sswv-policy.yml
$ kubectl -f ~/leonidas-sswv-policy.yml
```

We're now ready to rock'n'roll. From the executor's perspective at least…

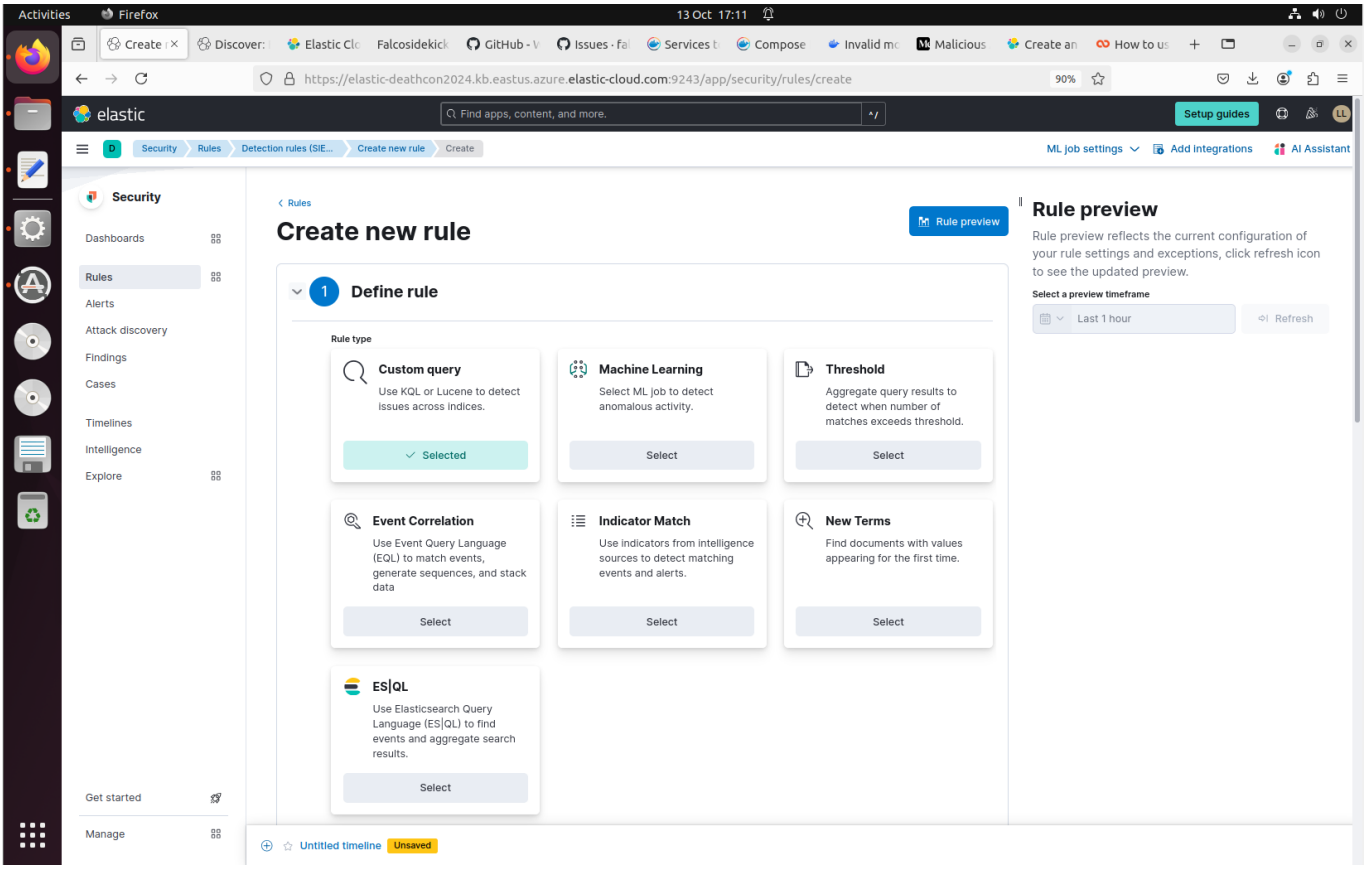
Let's prepare our detection stack as well, by loading the detections for these new test cases into the SIEM.

6. Generate the Sigma rules and corresponding Lucene queries for the new definitions

```
$ cd generator
$ poetry run ./generator.py sigma
$ cd output/leonidas
$ for d in `grep -ril dero  ../sigma/`; do poetry run sigma convert  -t lucene -p ecs_kubernetes $d;
done
Parsing Sigma rules  [####################################]  100%
kubernetes.audit.kind:Event AND (kubernetes.audit.verb:delete AND
kubernetes.audit.objectRef.resource:daemonsets)
Parsing Sigma rules  [####################################]  100%
kubernetes.audit.kind:Event AND (kubernetes.audit.verb:delete AND
kubernetes.audit.objectRef.resource:deployments)
Parsing Sigma rules  [####################################]  100%
kubernetes.audit.kind:Event AND (kubernetes.audit.verb:create AND
kubernetes.audit.objectRef.apiGroup:apps AND kubernetes.audit.objectRef.resource:daemonset)
```

7. Arm Elastic: Create Alerts with the queries generated above

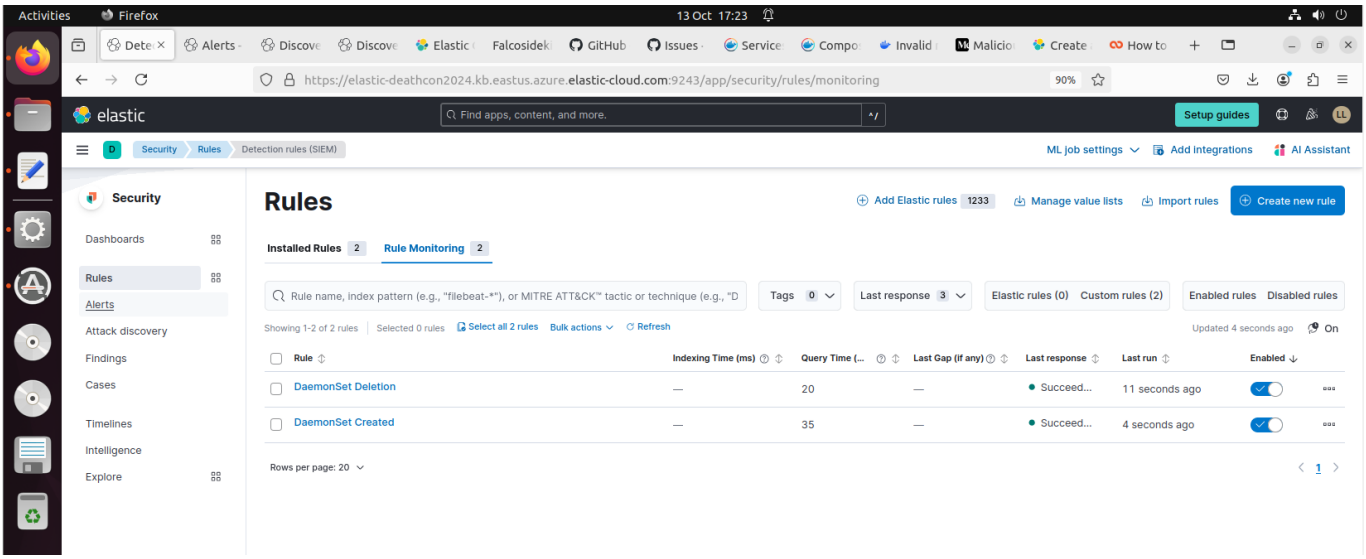
From the home page, go to Security > select Rules from the sidebar > Detection Rules (SIEM) > click the “Create new Rule” button on the top right.



In the new window, populate the fields as below:

- Rule Type: “Custom query”
- Source: (Index Patterns) leave only “logs-*” selected
- Custom query: the Lucene query from the `sigma convert` output
- Set a Name, Description and Severity
- Set a schedule
- And no actions

Finally, hit “Create and Enable”. Do this as many times as you prefer, to experiment with different alert queries:



8. Arm Falco: Define a Falco rule to detect cronjob additions

To exercise also the container/node-level detection capabilities of the house, we can also set up a Falco rule which would detect one of the SSWW TTPs. The last test case, in particular, serves as the only candidate, as setting a cron job for persistence is a host-based technique (in contrast to the other control plane TTPs) and a well-known one, historically, for establishing persistence. Additionally, it's an activity which is very unlikely to be observed for legitimate purposes in a Kubernetes environment, given the "containerised" alternatives offered by the platform.

So let's go ahead and define a custom Falco rule, focusing on this TTP:

Why Custom? Surely there's a pre-made Falco rule for Cron?

Detection in Kubernetes and generally containerised environments presents a lot of caveats. Another one of them was identified through painful trial and error while preparing this workshop guide. There is in fact, an incubating Falco rule for detecting Cron jobs scheduled… However, it would not capture our attack

Move the file `falco-detect-cron.yaml` under your VM's `/etc/falco/rules.d/`. This should not need a restart of the service to be loaded, but check for any errors just in case:

```
$ sudo journalctl -u falco-modern-bpf.service -f
```

We are now all set to proceed to the execution.

To recap, we should have all of the following systems up and running:

Minikube	Our cluster is running, optionally with the dashboard UI accessible in the browser
Leonidas	Leonidas is deployed and running, with its service forwarded to localhost's port 5000
Falco	Falco, falcosidekick and optionally falcosidekick UI – all containers running
Elastic	Audit Logs & Falco events are all being forwarded into Elastic
Jupyter	The Jupyter Notebook service running, accessible on port 8888

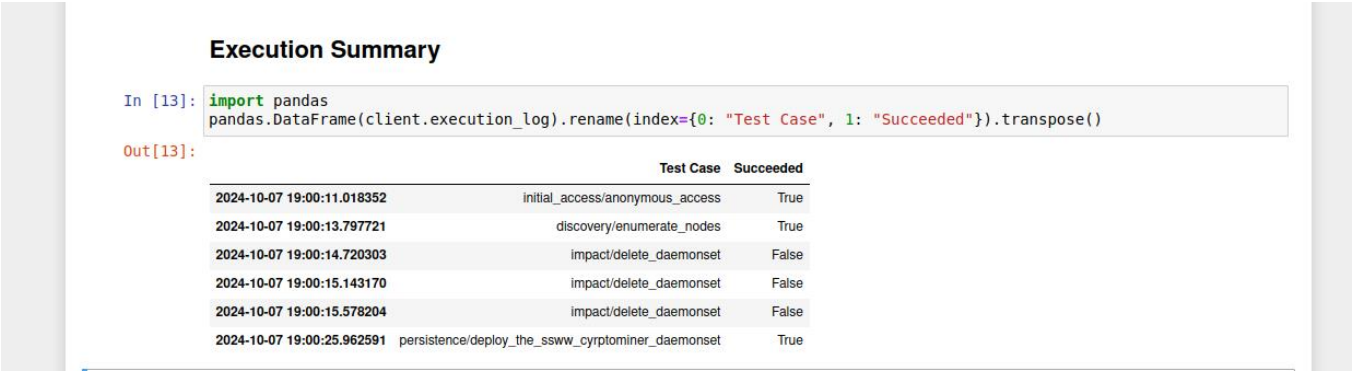
Without further ado, let's get the emulation underway:

9. Click-through all steps of Jupyter notebook

Select the first cell, and press the "Run" button.

Once completed, repeat for every next one.

After executing the last cell, if all went according to plan, your printed summary table should look like this:



Execution Summary

```
In [13]: import pandas
         pandas.DataFrame(client.execution_log).rename(index={0: "Test Case", 1: "Succeeded"}).transpose()
```

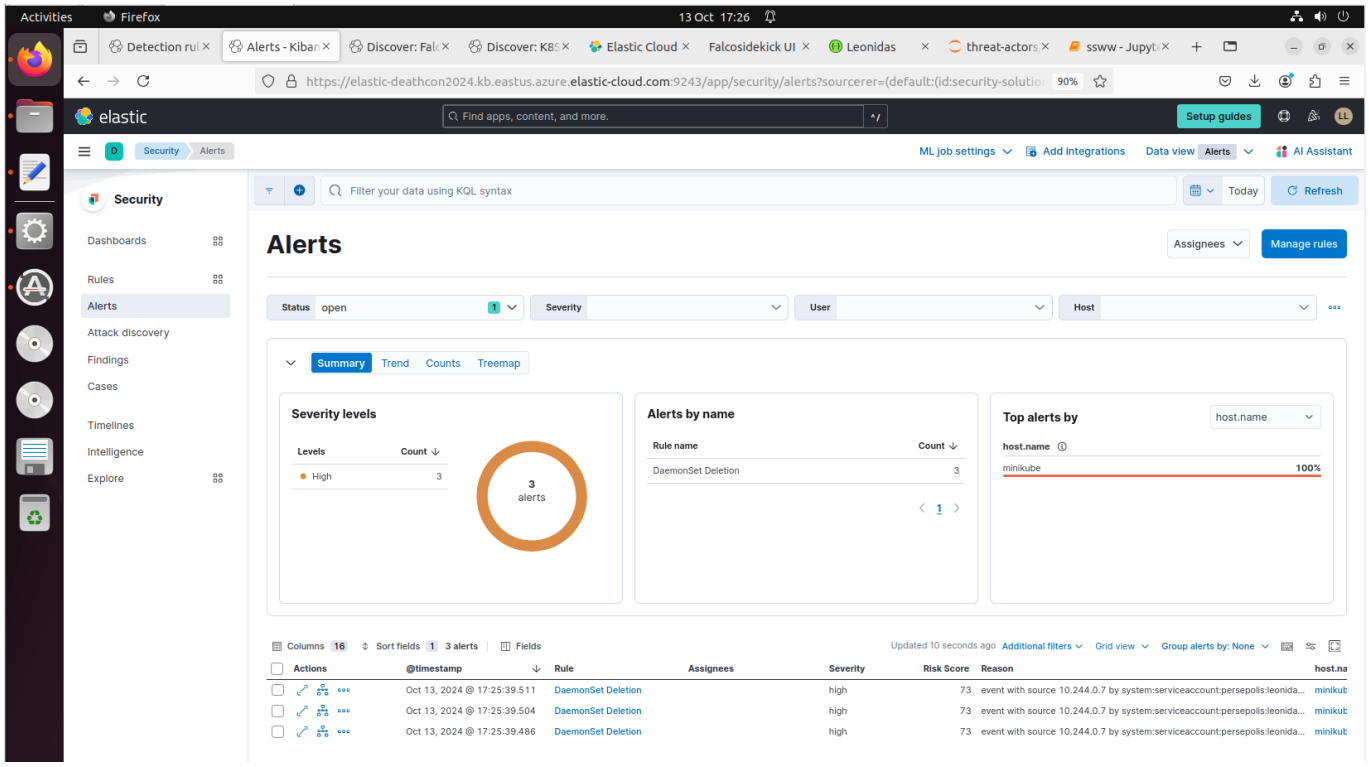
Out[13]:

	Test Case	Succeeded
2024-10-07 19:00:11.018352	initial_access/anonymous_access	True
2024-10-07 19:00:13.797721	discovery/enumerate_nodes	True
2024-10-07 19:00:14.720303	impact/delete_daemonset	False
2024-10-07 19:00:15.143170	impact/delete_daemonset	False
2024-10-07 19:00:15.578204	impact/delete_daemonset	False
2024-10-07 19:00:25.962591	persistence/deploy_the_ssww_cyrptominer_daemonset	True

Validating Detection Capability

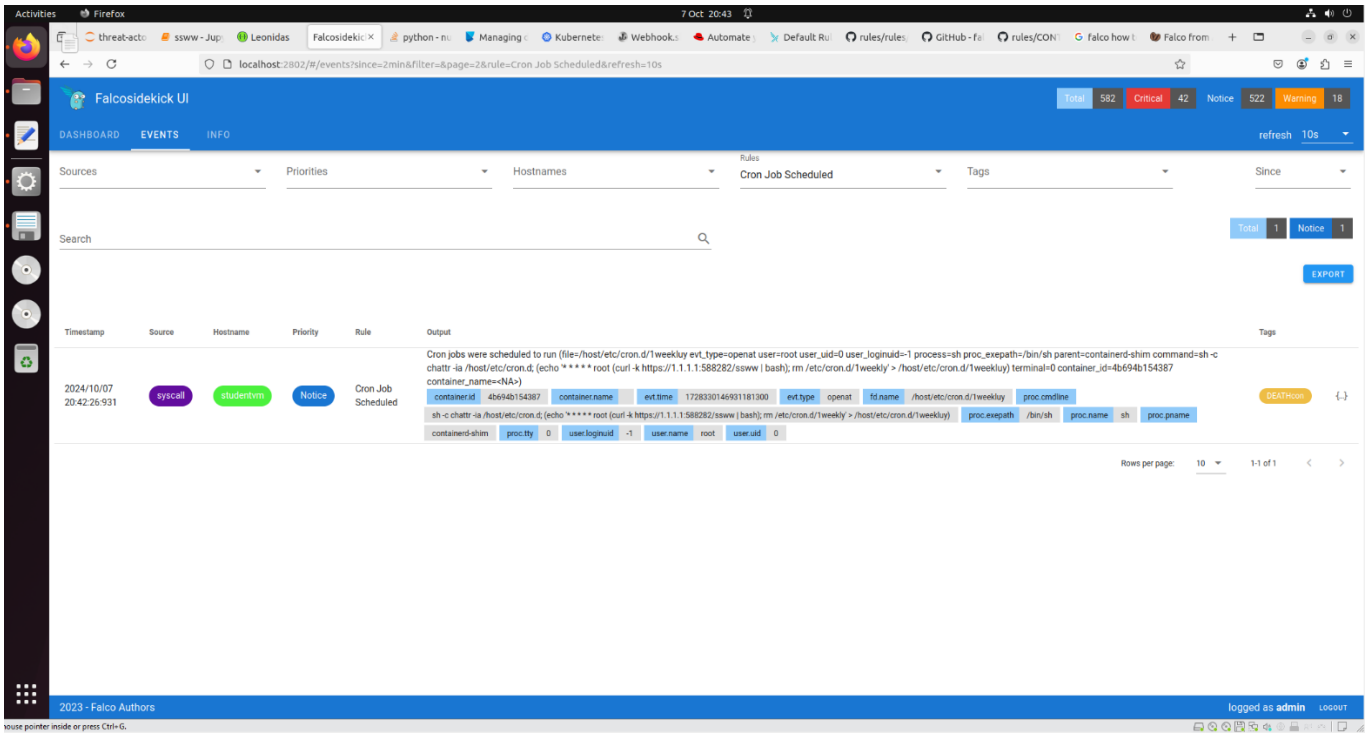
1. Check Elastic for Alerts triggered

To confirm our setup, refreshing the Alerts tab after a few seconds, we should see a few alerts raised as a result of our SSWW simulation



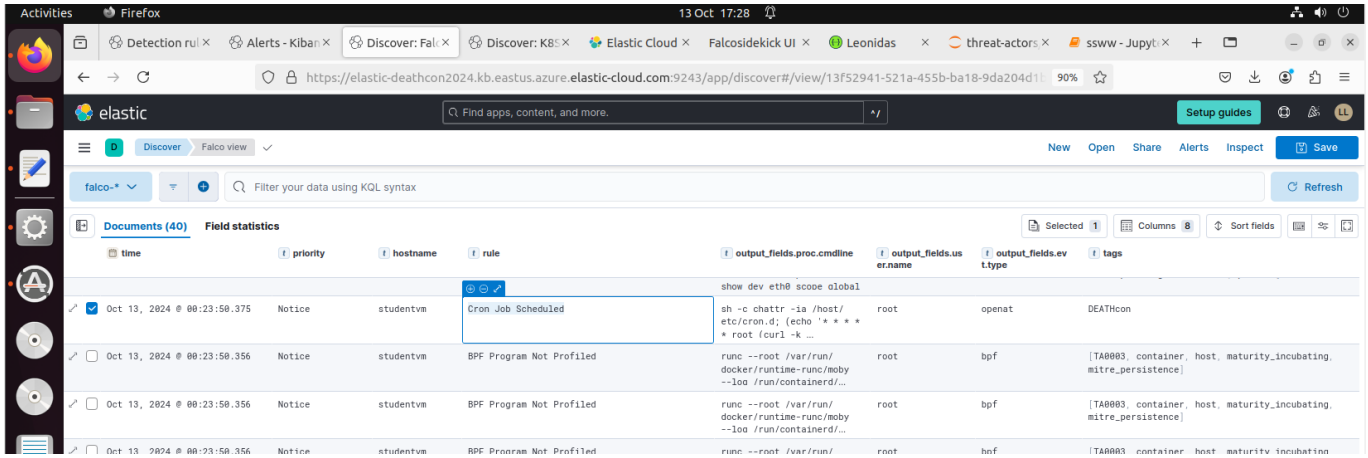
2. Check Falco for Alerts triggered

The SSWW activity should also trigger our kernel-based Cron Addition rule. Check the falcosidekick UI to verify



3. Check Elastic > Falco view for new documents

And to confirm Falco alert signals are ingested into Elastic, refresh the View for your Falco index



2 Exercise – RBAC Buster

RBAC buster was a popular campaign targeting Kubernetes environments, that leveraged weak RBAC permissions to establish privileged persistence inside a cluster.

Repeat the entire flow we went through in this workshop, to create another emulation plan for the RBAC buster campaign. Note that you can combine the existing “Create Service Account” test case, with a new “Create ClusterRoleBinding” one, that will bind the attacker’s SA to the cluster-admin role.

Once you execute them inside the cluster, switch to the SIEM, and engineer detections for those activities to complete the YAMLs.

Put together the new YAML definitions (including the logic for detections) and the new playbook, and submit a pull request to the upstream Leonidas repository for the new content. Alternatively, you can write a short public blog post about it. If you enjoyed using Leonidas for Kubernetes, speaking about it this way could help us spread the word about it, and help blue teams around the world.

3 Closing Notes

Even with a “silent” policy and a small scale cluster like ours, Kubernetes Audit logs can acquire substantial volume and eat up gigabytes of space. So don’t forget to **stop the accumulation of logs into the shared Elastic stack**, as a sign of kindness to the organisers and your fellow students. This can be achieved either by removing your Elastic indexes, or by turning off your cluster and falco instance, once done with this workshop:

```
$ minikube stop
$ docker compose stop
```

Finally, as a living open-source project, Leonidas is not immune to bugs and errors in the code. If you spot any issues during your testing, kindly raise it on GitHub! Pull requests will also be greatly appreciated!

Appendix I - Troubleshooting

Issue: Minikube API Server Failed to Start?

This is most likely due to resource requirements.

1. Clean up some Docker leftovers, free up disk space

```
$ docker system prune
```

2. Power cycle Minikube

```
$ minikube delete --all=true --purge=true
$ minikube start --driver=docker --memory=max --cpus=max -v=2

# Ensure all components “Running”
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

# Re-Download kubectl
$ minikube kubectl
```

And depending on your progress, repeat setup steps (Workshop Boilerplate):

```
# Re-Configure Audit Policy
$ mkdir -p ~/.minikube/files/etc/ssl/certs
$ cp ~/student-materials/audit-policy.yaml ~/.minikube/files/etc/ssl/certs
$ docker exec minikube touch /var/log/audit.log

$ docker cp minikube:/etc/kubernetes/manifests/kube-apiserver.yaml  .
$ patch kube-apiserver.yaml audit.patch

$ minikube stop
$ docker cp kube-apiserver.yaml minikube:/etc/kubernetes/manifests/kube-apiserver.yaml
$ minikube start --driver=docker --memory=max --cpus=max -v=2 \
--extra-config=apiserver.audit-policy-file=/etc/ssl/certs/audit-policy.yaml \
--extra-config=apiserver.audit-log-path=/var/log/audit.log

# Forward Audit Logs to Elastic
$ kubectl apply -f elastic-agent-managed-kubernetes.yml

# Re-Deploy Leonidas
$ cd leonidas/generator
$ poetry run ./generator.py build-image
$ LEOPOD=$(kubectl get pods -A -l app=leonidas -o jsonpath='{.items[*].metadata.name}')
$ LEONS=$(kubectl get pods -A -l app=leonidas -o jsonpath='{.items[*].metadata.namespace}')
$ kubectl delete pod $LEOPOD -n $LEONS
$ kubectl port-forward deployment/leonidas-deployment 5000:5000 -n $LEONS

# Restart Falco
$ cd ~
$ docker compose up --detach
```

Issue: Leonidas Pod Failed to Start?

As every Kubernetes engineer can confirm, `CrashLoopBackoff` is the most common, and most painful error to debug, due to how non-descriptive it can be, and the amount of different reasons resulting to it. In our case however there are 3 issues which are more likely:

1. `ImagePull*` error – effectively, the Kubelet can't find the Leonidas image in the public repository (ttl.sh, if you followed guidance) that you specified in your `config.yaml`. Check the lifespan tag you set and how much time has elapsed since you last pushed (through `generator.py build-image`, if you followed guidance), repeating the build process once again if needed, before re-deploying - whether re-applying all of the Leonidas resources, or simply deleting the Leonidas pod to trigger a re-deploy

2. Resource Exhaustion – Play around for a while and you'll be creating too many Kubernetes objects than what the small minikube cluster can handle. See the previous Issue FAQ for tips on how troubleshooting this and generally, clean up some space.

3. A bug in the Python code would mean that the `python leonidas.py` entrypoint command has exited. So check what modifications you have introduced to the Python code (hint: `git diff HEAD`) and `git checkout` to a working version