

Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого

—
Институт компьютерных наук и кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 2

«Мониторинг кластера при помощи Falco»

по дисциплине «Программно-аппаратные средства обеспечения
информационной безопасности»

Выполнила

студентка гр. 5131001/10302

Куковьякина Д. А.

<подпись>

Преподаватель

Малышев Е.

В.

<подпись>

Санкт-Петербург

2025

Цель работы

Получение навыков мониторинга кластера при помощи Falco.

Формулировка задания

В рамках выполнения работы необходимо выполнить следующие задачи:

- Развернуть локальный кластер k8s с использованием minikube
- Настроить рабочее окружение (kubectl, helm)
- Изучить базовые команды для управления кластером
- Установить и сконфигурировать Falco при помощи Helm-чарта
- Настроить оповещения о событиях безопасности в телеграм чат
- Инициировать событие безопасности и заставить сработать алерт
- Создать собственное правило и подготовить его демонстрацию

Ход работы

Подготовка окружения

Архитектура Kubernetes состоит из двух основных частей: плоскости управления (control plane) и рабочих узлов (worker nodes).

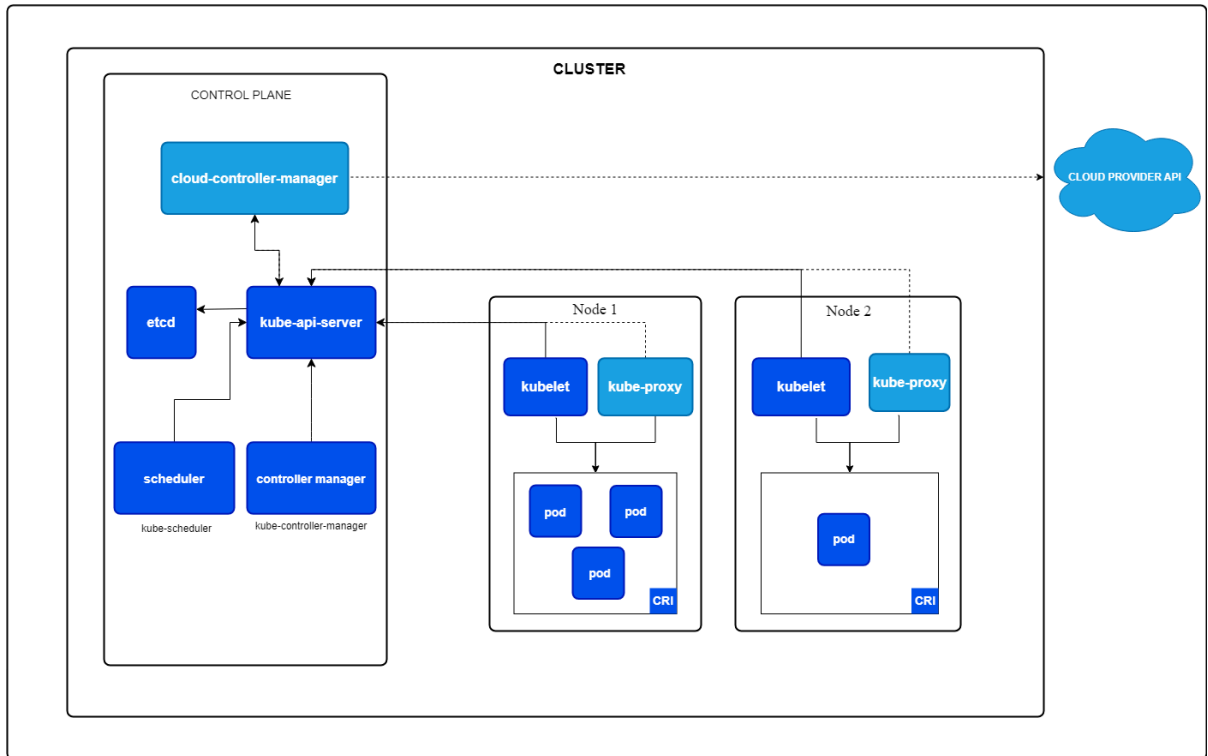


Рисунок 1 – Архитектура kubernetes

Система управления:

- kube-apiserver предоставляет интерфейс для взаимодействия с кластером Kubernetes.
- Etcd – распределенное хранилище данных, в котором хранятся все данные кластера.
- kube-scheduler – планировщик, который определяет, на каком узле запускать контейнеры.
- kube-controller-manager управляет различными контроллерами, обеспечивающими желаемое состояние кластера.
- cloud-controller-manager интегрирует Kubernetes с облачными провайдерами (если используется облачная среда).

Рабочие узлы:

- kubelet – агент, который работает на каждом узле и управляет контейнерами на этом узле.
- kube-proxy – сетевой прокси, который обеспечивает сетевое взаимодействие между контейнерами и внешним миром.
- container runtime – среда выполнения контейнеров, такая как Docker или containerd, которая запускает контейнеры.

Основные объекты Kubernetes:

- Node — это рабочая машина в кластере Kubernetes, которая выполняет Pod-ы. Это может быть физический или виртуальный сервер.
- Pod – наименьшая единица развертывания в Kubernetes. Он представляет собой группу из одного или нескольких контейнеров (например, Docker), которые разделяют общее хранилище, сеть и спецификации запуска.
- Deployment – объект, который управляет репликами Pod-ов и обеспечивает их обновление. Он является одним из контроллеров, который выполняется в kube-controller-manager.
- Service – объект, который обеспечивает сетевой доступ к Pod-ам.
- Namespace – объект, который позволяет разделить кластер на логические части.

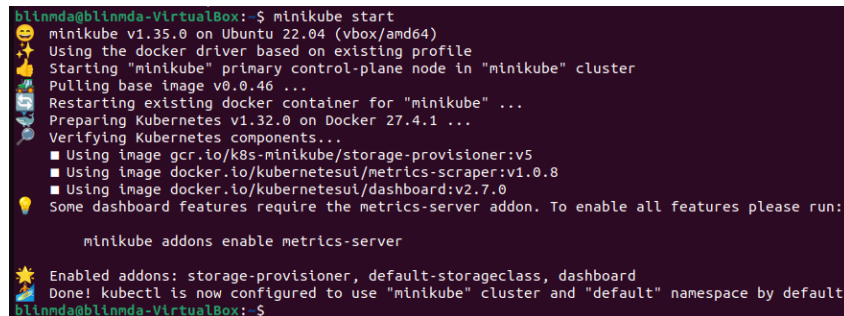
Docker – это платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры позволяют упаковать приложение со всеми его зависимостями в единый блок, который можно легко переносить между различными средами. Kubernetes использует контейнеры для развертывания приложений. Он управляет жизненным циклом контейнеров Docker, обеспечивая их запуск, масштабирование и обновление.

Minikube – это инструмент, который позволяет запускать Kubernetes локально на компьютере. Он создает одноузловой кластер Kubernetes внутри виртуальной машины. Minikube полезен для разработчиков, которые хотят протестировать свои приложения Kubernetes локально перед развертыванием в производственной среде.

Kubectl – это инструмент командной строки для управления кластерами Kubernetes. Он позволяет выполнять различные операции, такие как развертывание приложений, масштабирование приложений, проверка состояния кластера и т. д. Kubectl является основным инструментом для взаимодействия с Kubernetes.

Helm – это менеджер пакетов для Kubernetes. Он позволяет упаковывать и развертывать приложения Kubernetes как единые пакеты, называемые чартами. Helm упрощает развертывание и управление сложными приложениями и позволяет автоматизировать установку и обновление приложений на кластере Kubernetes.

Установим данные утилиты и запустим minikube (рисунок 2).



```
blinmda@blinmda-VirtualBox:~$ minikube start
minikube v1.35.0 on Ubuntu 22.04 (vbox/amd64)
Using the docker driver based on existing profile
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.46 ...
Restarting existing docker container for "minikube" ...
Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
Verifying Kubernetes components...
  ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
  ■ Using image docker.io/kubernetes/metrics-scraper:v1.0.8
  ■ Using image docker.io/kubernetes/dashboard:v2.7.0
  ⚠ Some dashboard features require the metrics-server addon. To enable all features please run:

      minikube addons enable metrics-server

  ⚡ Enabled addons: storage-provisioner, default-storageclass, dashboard
  🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
blinmda@blinmda-VirtualBox:~$
```

Рисунок 2 – Запуск minikube

Namespace в Kubernetes — это способ разделения ресурсов кластера между несколькими командами, проектами или средами (например, разработка, тестирование, продакшн). Namespaces обеспечивают изоляцию на уровне кластера Kubernetes.

После создания namespace развернем приложение (рисунок 3). Оно содержит:

- ubuntu-deployment – создает контейнер на основе образа Ubuntu, и контейнер будет находиться в состоянии сна.
- foo-app – запускает простой эхо-сервер.
- foo-service – предоставляет доступ к Pod foo-app через порт 8080.
- nginx-deployment – запускает веб-сервер Nginx.
- bar-app – запускает простой эхо-сервер.
- bar-service – предоставляет доступ к Pod bar-app через порт 8080.

- example-ingress – маршрутизирует HTTP-трафик на основе путей к соответствующим сервисам.

```
blinmda@blinmda-VirtualBox:~$ kubectl create namespace secured-ns
namespace/secured-ns created
blinmda@blinmda-VirtualBox:~$ kubectl -n secured-ns apply -f /home/blinmda/Desktop/lab/falco_stub/deployment.yaml
deployment.apps/ubuntu-deployment created
pod/foo-app created
deployment.apps/nginx-deployment created
service/foo-service created
pod/bar-app created
service/bar-service created
ingress.networking.k8s.io/example-ingress created
blinmda@blinmda-VirtualBox:~$
```

Рисунок 3 – Развертка приложения

На рисунке 4 представлены запущенные поды в minikube dashboard.

Pods				
Name	Images	Labels	Node	Status
bar-app	kicbase/echo-server:1.0	app: bar	minikube	Running
foo-app	kicbase/echo-server:1.0	app: foo	minikube	Running
nginx-deployment-647677fc66-4jxpd	nginx:1.14.2	app: nginx pod-template-hash: 647677fc66	minikube	Running
ubuntu-deployment-5fccb98689-zmn4t	ubuntu	app: ubuntu pod-template-hash: 5fccb98689	minikube	Running

Рисунок 4 – Запущенные поды

Настройка Falco

Falco — это инструмент безопасности с открытым исходным кодом, предназначенный для обнаружения аномального поведения в контейнерах и Kubernetes. Он отслеживает системные вызовы ядра и использует правила для выявления потенциальных угроз безопасности. Его архитектура представлена на рисунке 5.

Falco extended architecture

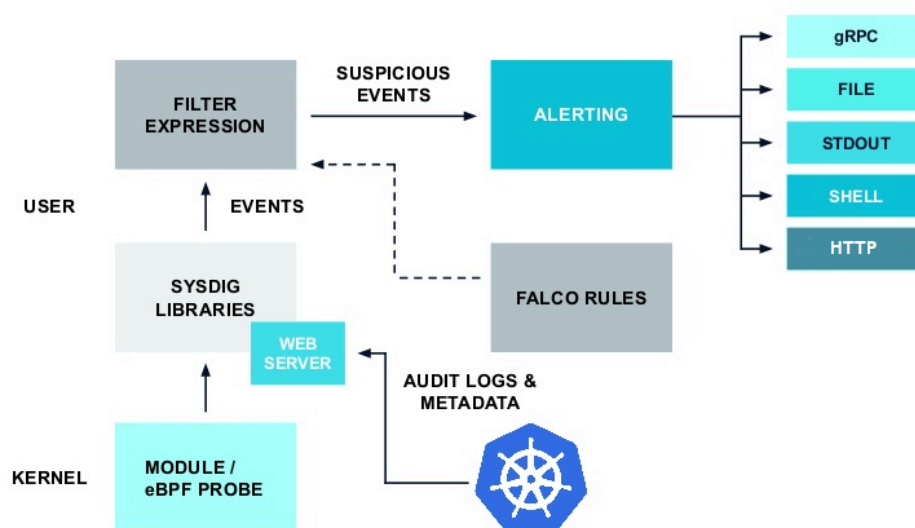


Рисунок 5 – Архитектура Falco

Установим Falco при помощи helm (рисунок 6).

```
helm repo add falcosecurity https://falcosecurity.github.io/charts
```

Эта команда добавляет репозиторий Helm под названием falcosecurity в локальный список репозитория. Helm использует репозитории для поиска и установки приложений в Kubernetes.

```
helm repo update
```

Эта команда обновляет локальный кэш репозитория Helm. Она загружает последние обновления из всех добавленных репозитория, включая falcosecurity.

```
helm pull falcosecurity/falco --untar
```

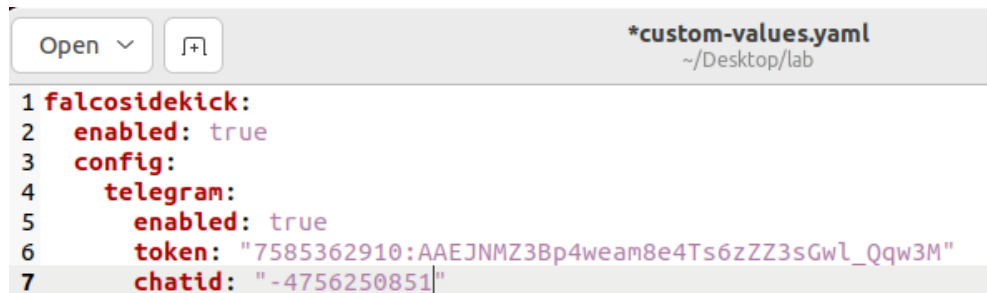
Эта команда загружает чарт Helm Falco из репозитория falcosecurity. В результате в текущем каталоге создается директория с файлами чарта Falco. Это позволяет локально изучить, и изменить файлы чарта, до установки.

```
blinmda@blinmda-VirtualBox:~$ helm repo add falcosecurity https://falcosecurity.github.io/charts
"falcosecurity" has been added to your repositories
blinmda@blinmda-VirtualBox:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "falcosecurity" chart repository
Update Complete. ☺Happy Helming!☺
blinmda@blinmda-VirtualBox:~$ helm pull falcosecurity/falco --untar
```

Рисунок 6 – Установка Falco

Далее необходимо настроить отправку уведомлений в телеграм. Falcosidekick — это инструмент, который помогает Falco взаимодействовать с различными системами для отправки оповещений о безопасности. Он действует как мост, соединяя обнаружения Falco с различными сторонними инструментами.

Для настройки необходимо задать токен бота и идентификатор чата, в который он добавлен (рисунок 7).

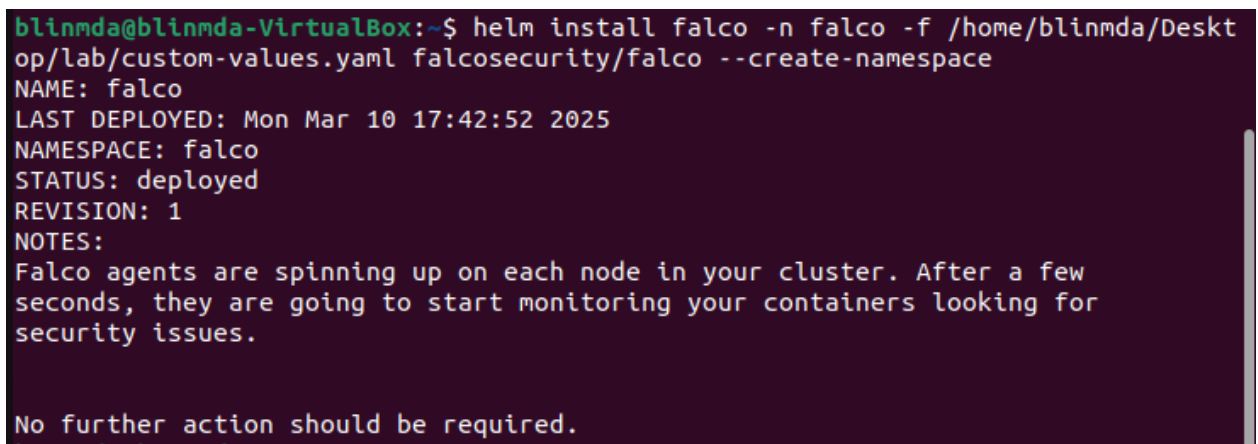


```
1 falcosidekick:
2   enabled: true
3   config:
4     telegram:
5       enabled: true
6       token: "7585362910:AAEJNMZ3Bp4weam8e4Ts6zZZ3sGwL_Qqw3M"
7       chatid: "-4756250851"
```

Рисунок 7 – Задание переменных

После этого установим Falco командой (рисунок 5).

```
helm install falco -n falco falcosecurity/falco --create-namespace -f
custom-values.yaml
```



```
blinmda@blinmda-VirtualBox:~$ helm install falco -n falco -f /home/blinmda/Desktop/lab/custom-values.yaml falcosecurity/falco --create-namespace
NAME: falco
LAST DEPLOYED: Mon Mar 10 17:42:52 2025
NAMESPACE: falco
STATUS: deployed
REVISION: 1
NOTES:
Falco agents are spinning up on each node in your cluster. After a few
seconds, they are going to start monitoring your containers looking for
security issues.

No further action should be required.
```

Рисунок 8 – Установка Falco

Далее проверим, что все работает. Для этого зайдём в под ubuntu и посмотрим файл в директории etc (рисунок 9). Сработает уведомление (рисунок 10).

```

blinmda@blinmda-VirtualBox:~$ kubectl get pods -n secured-ns
NAME                                READY   STATUS    RESTARTS   AGE
bar-app                             1/1     Running   1 (51m ago) 70m
foo-app                             1/1     Running   1 (51m ago) 70m
nginx-deployment-647677fc66-4jxpd   1/1     Running   1 (51m ago) 70m
ubuntu-deployment-5fccb98689-zmn4t  1/1     Running   1 (51m ago) 70m
blinmda@blinmda-VirtualBox:~$ kubectl exec -it -n secured-ns ubuntu-deployment-5fccb98689-zmn4t -- bash
root@ubuntu-deployment-5fccb98689-zmn4t:/# cat /etc/pam.d/passwd
#
# The PAM configuration file for the Shadow 'passwd' service
#
@include common-password
root@ubuntu-deployment-5fccb98689-zmn4t:/#

```

Рисунок 9 – Инициирование уведомления

```

falco_alert
3 участника

[Falco] [Warning] Read sensitive file untrusted

• Time: 2025-03-10 15:09:53.526225612 +0000 UTC
• Source: syscall
• Hostname: minikube
• Tags: T1555 container filesystem host maturity_stable
  mitre_credential_access
• Fields:
  • container.id: dc464dda0dc9
  • container.image.repository: ubuntu
  • container.image.tag: latest
  • container.name: k8s_ubuntu_ubuntu-deployment-5fccb98689-
    zmn4t_secured-ns_5b7d4902-f040-48af-8031-83e81b647753_1
  • evt.time: 1741619393526225612
  • evt.type: openat
  • fd.name: /etc/pam.d/passwd
  • k8s.ns.name: <nil>
  • k8s.pod.name: <nil>
  • proc.aname[2]: <nil>
  • proc.aname[3]: <nil>
  • proc.aname[4]: <nil>
  • proc.cmdline: cat /etc/pam.d/passwd
  • proc.exepath: /usr/bin/cat
  • proc.name: cat
  • proc.pname: bash
  • proc.tty: 34816
  • user.loginuid: 1
  • user.name: root
  • user.uid: 0

Output: 15:09:53.526225612: Warning Sensitive file opened for
reading by non-trusted program (file=/etc/pam.d/passwd
gparent=<NA> gpparent=<NA> gggparent=<NA> evt_type=openat
user=root user_uid=0 user_loginuid=1 process=cat
proc_exepath=/usr/bin/cat parent=bash command=cat
/etc/pam.d/passwd terminal=34816 container_id=dc464dda0dc9
container_image=ubuntu container_image_tag=latest
container_name=k8s_ubuntu_ubuntu-deployment-5fccb98689-

```

Рисунок 10 – Уведомление

Правила

Для получения информации о том, в каком namespace произошло событие воспользуемся k8s-metacollector. Он предназначен для сбора и метаданных Kubernetes, которые обогащают события безопасности, генерируемые Falco. Его поле k8smeta.ns.name содержит имя namespace, в котором произошло событие. Для включения данного сборщика используем опцию `--set collectors.kubernetes.enabled=true`.

Добавим в дефолтные правила Falco проверку данного поля (рисунок 11) на соответствие namespace prod.

```
customRules:
  custom-default.yaml: |-
    - macro: ns_name
      condition: k8smeta.ns.name = "prod"
```

Рисунок 11 – Проверка namespace

Загрузим новые правила как собственные, при этом отключив использование дефолтного файла falco_rules.yaml.

Далее было создано собственное правило (рисунок 12). Оно отслеживает открытие файлов для записи в директории test и присылает об этом уведомления (рисунок 13).

```
1 customRules:
2   custom-rules.yaml: |-
3     - rule: Write below test directory
4       desc: An attempt to write to /test directory (custom rule)
5       condition: >
6         (evt.type in (open,openat,openat2) and evt.is_open_write=true and fd.typechar='f' and
7         fd.num>=0)
8         and fd.name startswith /test
9       output: "File below /test opened for writing (file=%fd.name pcmdline=%proc.pcmdline
10      gparent=%proc.aname[2] ggparent=%proc.aname[3] gggparent=%proc.aname[4] evt_type=%evt.type
11      user=%user.name user_uid=%user.uid user_loginuid=%user.loginuid process=%proc.name
12      proc_exepath=%proc.exepath parent=%proc.pname command=%proc.cmdline terminal=%proc.tty
13      %container.info)"
14     priority: WARNING
15     tags: [filesystem]
```

Рисунок 12 – Новое правило

```
[Falco] [Warning] Write below test directory
• Time: 2025-03-10 18:10:44.215881144 +0000 UTC
• Source: syscall
• Hostname: minikube
• Tags: filesystem
• Fields:
  • container.id: 70b77a8d28c7
  • container.image.repository: ubuntu
  • container.image.tag: latest
  • container.name: k8s_ubuntu_ubuntu-deployment-5fccb98689-
vrzg_prod_04d1640a-bf11-4481-9d02-6fe78aa67c1f_1
  • evt.time: 1741630244.215881144
  • evt.type: openat
  • fd.name: /test/testfile
  • k8s.ns.name: <nil>
  • k8s.pod.name: <nil>
  • proc.cmdline: touch /test/testfile
  • proc.exepath: /usr/bin/touch
  • proc.name: touch
  • proc.pcmdline: <NA>
  • proc.pname: <NA>
  • proc.tty: 34816
  • user.loginuid: 1
  • user.name: root
  • user.uid: 0

Output: 18:10:44.215881144: Warning Custom rule: File below /test
opened for writing (file=/test/testfile pcmdline=<NA>
evt_type=openat user=root user_uid=0 user_loginuid=1
process=touch proc_exepath=/usr/bin/touch parent=<NA>
command=touch /test/testfile terminal=34816
container_id=70b77a8d28c7 container_image=ubuntu
container_image_tag=latest container_name=k8s_ubuntu_ubuntu-
deployment-5fccb98689-vrzg_prod_04d1640a-
bf11-4481-9d02-6fe78aa67c1f_1 k8s_ns=<NA> k8s_pod_name=<NA>)
```

Рисунок 13 – Уведомление по новому правилу

Тестирование

Рассмотрим несколько правил и проверим, что они работают.

Drop and execute new binary in container. Описание правила представлено на рисунке 14.

Это правило предназначено для обнаружения подозрительных процессов, запущенных внутри контейнера, которые не являются частью базового образа контейнера. Оно нацелено на обнаружение вредоносной активности, когда злоумышленник загружает и запускает исполняемый файл внутри контейнера после получения первоначального доступа.

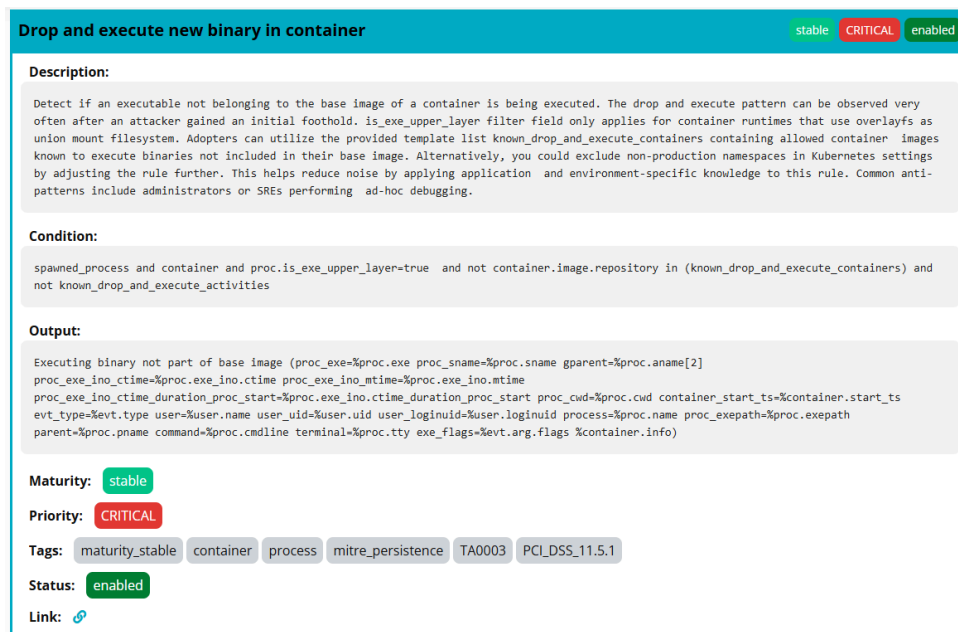


Рисунок 14 - Drop and execute new binary in container

Разберем правило:

- `spawned_process` – когда внутри контейнера запускается новый процесс.
- `container` – событие происходит внутри контейнера.
- `proc.is_exe_upper_layer=true` – исполняемый файл, который запускается, находится в верхнем слое файловой системы контейнера. В контексте Docker и других контейнерных сред, использующих overlays, это означает, что файл не является частью базового образа контейнера, а был добавлен позже.
- `not container.image.repository in (known_drop_and_execute_containers)` –исключает контейнеры, которые

известны тем, что они намеренно запускают исполняемые файлы, не входящие в их базовые образы.

- `not known_drop_and_execute_activities` – исключает известные действия "сбросить и выполнить".

В правилах также часто используется поле `evt.dir`. Когда приложение или процесс выполняет системный вызов, этот вызов может быть либо "входящим" (entering), либо "исходящим" (exiting).

"Входящий" системный вызов означает, что процесс только начал выполнение системного вызова (<).

"Исходящий" системный вызов означает, что процесс завершил выполнение системного вызова и возвращает результат (>).

На рисунке 15 представлено полученное уведомление. Для его срабатывания при помощи `art` был установлен и запущен `nmap`.

```
[Falco] [Critical] Drop and execute new binary in container
• Time: 2025-03-11 17:11:49.478946508 +0000 UTC
• Source: syscall
• Hostname: minikube
• Tags: PCI_DSS_11.5.1 TA0003 container maturity_stable
mitre_persistence process
• Fields:
  • container.id: 0b2089bcc334
  • container.image.repository: ubuntu
  • container.image.tag: latest
  • container.name: k8s_ubuntu_ubuntu-deployment-5fccb98689-
vrxyz_prod_04d1640a-bf11-4481-9d02-6fe78aa67c1f_6
  • container.start_ts: 1741711346260811022
  • evt.arg.flags: EXE_WRITABLE|EXE_UPPER_LAYER
  • evt.time: 1741713109478946508
  • evt.type: execve
  • k8s.ns.name: <nil>
  • k8s.pod.name: <nil>
  • proc.aname[2]: containerd-shim
  • proc.cmdline: nmap
  • proc.cwd: /
  • proc.exe: nmap
  • proc.exe_ino.ctime: 1741713103360295637
  • proc.exe_ino.ctime_duration_proc_start: 6118403044
  • proc.exe_ino.mtime: 1711954728000000000
  • proc.exepath: /usr/bin/nmap
  • proc.name: nmap
  • proc.pname: bash
  • proc.sname: bash
  • proc.tty: 34816
  • user.loginuid: -1
  • user.name: root
  • user.uid: 0

Output: 17:11:49.478946508: Critical Executing binary not part of
base image (proc_exe=nmap proc_sname=bash
gparent=containerd-shim
```

Рисунок 15 – Уведомление `nmap`

Следующее правило – `Packet socket created in container` (рисунок 16).

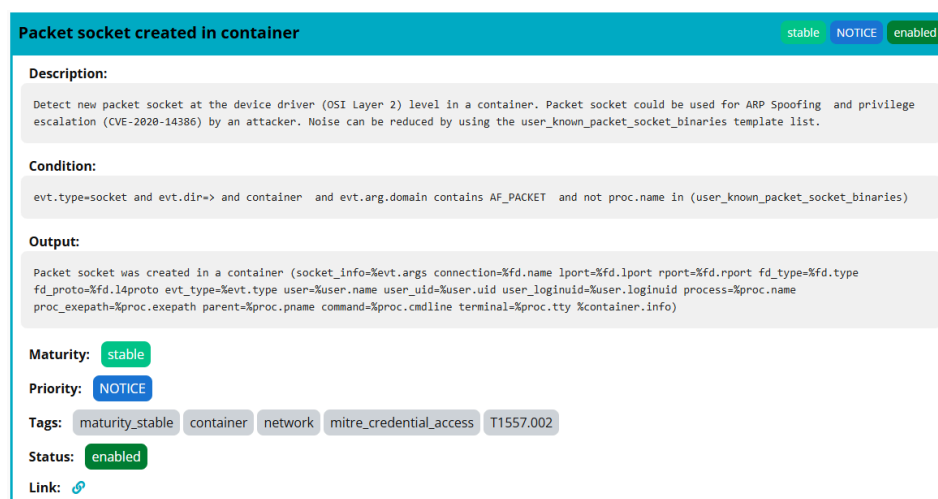


Рисунок 16 – Packet socket created in container

Это правило предназначено для обнаружения подозрительной сетевой активности внутри контейнеров, связанной с созданием пакетных сокетов. Пакетные сокет могут использоваться для атак, таких как ARP-спуфинг и эскалация привилегий (например, CVE-2020-14386), так как они позволяют приложениям отправлять и принимать необработанные сетевые пакеты (Ethernet-фреймы).

Разберем правило:

- `evt.type=socket` – срабатывает, когда происходит событие, связанное с созданием сокета.
- `evt.dir=>` – гарантирует, что правило проверяет тип домена сокета только после того, как системный вызов `socket` завершился и вернул результат.
- `container` – событие происходит внутри контейнера.
- `evt.arg.domain contains AF_PACKET` – проверяет, что тип домена сокета содержит `AF_PACKET`. `AF_PACKET` — это адресный домен сокета, который позволяет программе взаимодействовать с сетевым интерфейсом на уровне Layer 2 (уровень драйвера устройства).
- `not proc.name in (user_known_packet_socket_binaries)` – исключает процессы, которые известны тем, что они намеренно создают пакетные сокет.

Для срабатывания данного правила установим tcpdump и запустим его tcpdump -i any (рисунок 17).

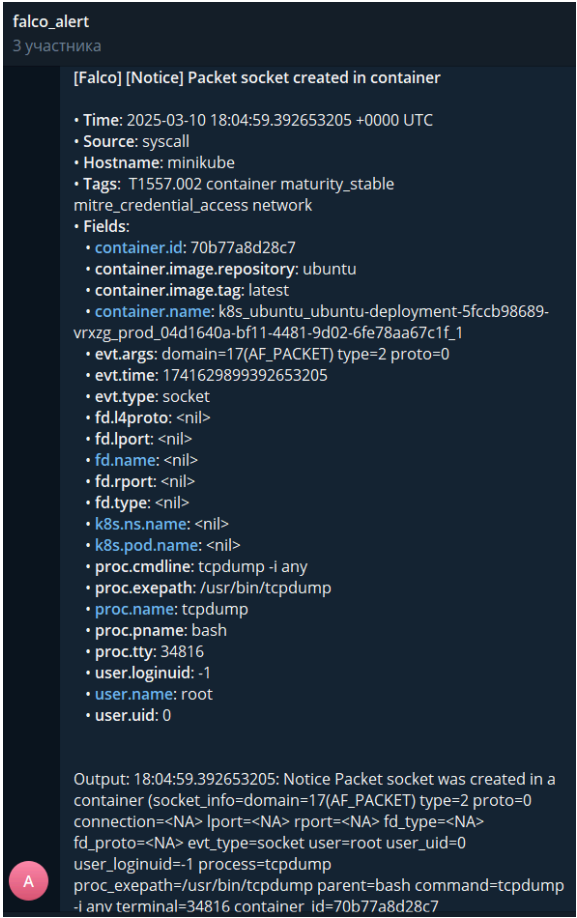


Рисунок 17 – Уведомление tcpdump

Следующее правило – Create Symlink Over Sensitive Files (рисунок 18).

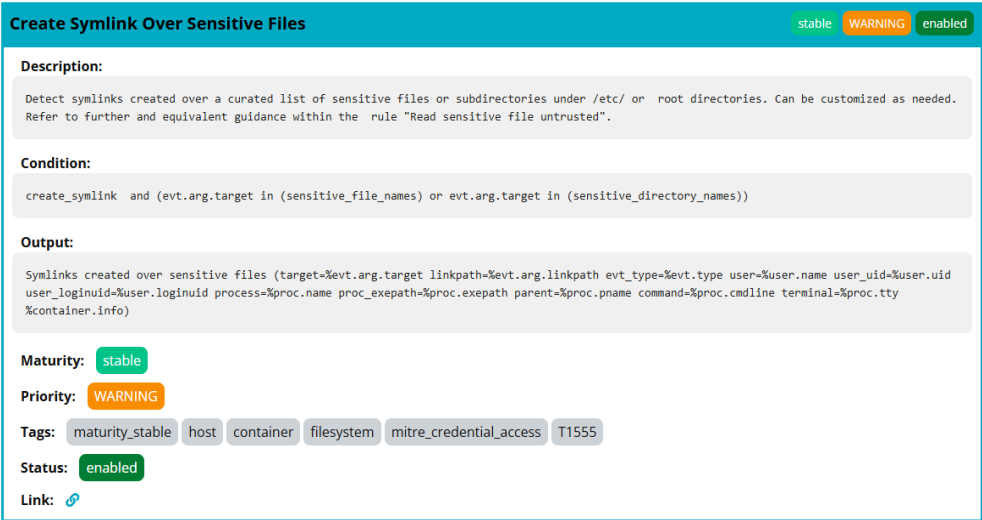


Рисунок 18 – Create Symlink Over Sensitive Files

Это правило предназначено для обнаружения подозрительных действий, связанных с созданием символических ссылок на критически важные файлы или каталоги. Например, злоумышленник может создать символическую ссылку, которая указывает на вредоносный файл, подменяя таким образом легитимный конфигурационный файл.

Разберем правило:

- `create_symlink` – срабатывает, когда происходит системный вызов, связанный с созданием символической ссылки (обычно `symlink` или `symlinkat`).

- `(evt.arg.target in (sensitive_file_names) or evt.arg.target in (sensitive_directory_names))` – проверяет, что целевой путь (`evt.arg.target`) символической ссылки находится в одном из двух списков:

`sensitive_file_names` (список чувствительных файлов), например `/etc/passwd`, `/etc/shadow`, `/etc/ssh/ssh_config`.

`sensitive_directory_names` (список чувствительных подкаталогов), например `/etc/ssh/`, `/etc/sudoers.d/`.

Для срабатывания правила введем команду `ln -s /etc/shadow /tmp/shadow_link`. Получим уведомление (рисунок 19).

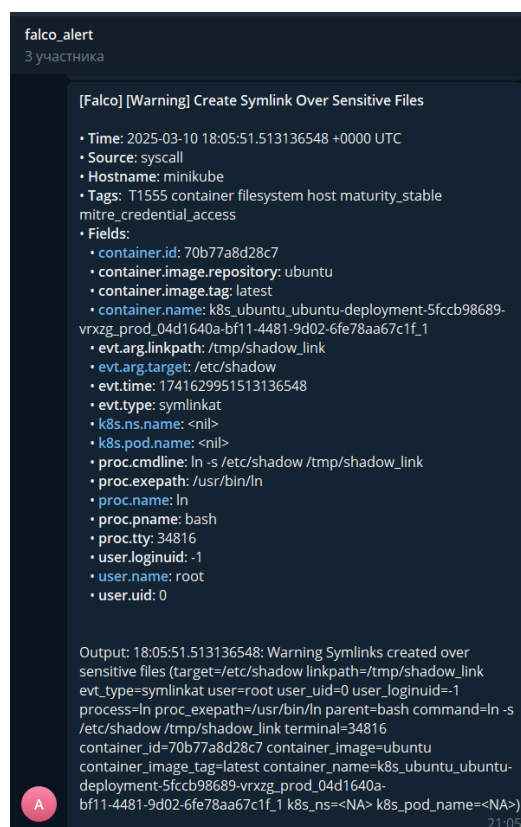


Рисунок 19 – Уведомление symlink

В целом правила Falco можно разделить на 3 основные категории:

1. Правила, связанные с файловой системой.

Обнаружение попыток чтения или записи в критические системные файлы (например, /etc/passwd, /etc/shadow). Используется для обнаружения несанкционированного доступа к учетным данным или конфигурациям.

Отслеживание попыток изменения исполняемых файлов внутри контейнера. Помогает обнаружить внедрение вредоносного кода.

Обнаружение создания символических ссылок на чувствительные файлы или каталоги. Предотвращает подмену конфигурационных файлов или обход ограничений доступа.

2. Правила, связанные с процессами.

Обнаружение запуска процессов, которые обычно не должны выполняться внутри контейнера (например, shell-скрипты, исполняемые файлы из /tmp).

Отслеживание запуска контейнеров с повышенными привилегиями. Предотвращает потенциальную эскалацию привилегий.

3. Правила, связанные с сетью:

Отслеживание необычных сетевых соединений (например, исходящие соединения на неизвестные IP-адреса или порты). Используется для обнаружения попыток эксфильтрации данных или установления обратного shell-a.

Обнаружение создания пакетных сокетов на уровне драйвера устройства. Предотвращает атаки, такие как ARP-спуфинг и эскалация привилегий.

Вывод

В ходе данной лабораторной работы был развернут одноранговый кластер minikube. В нем были созданы несколько namespaces, а также установлен Falco. Он был настроен на отслеживание событий, происходящих только в одном namespace, также было создано пользовательское правило.