O'REILLY®

Practical Cloud Native Security with Falco

Risk and Threat Detection for Containers, Kubernetes, and Cloud

Early Release

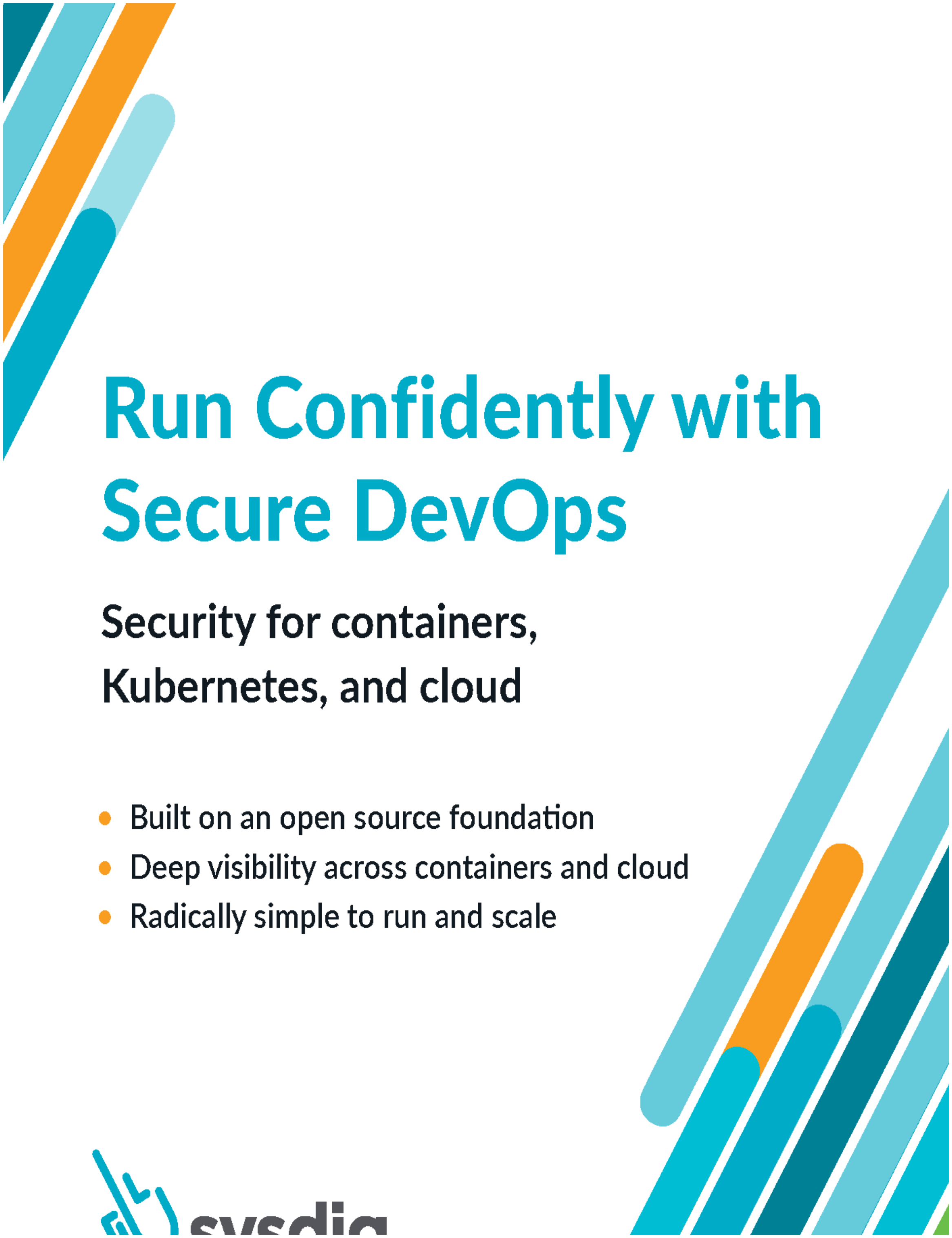
Raw & Unedited

Compliments of

sysdig

Loris Degioanni &
Leonardo Grasso

Sysdig



Run Confidently with Secure DevOps

Security for containers, Kubernetes, and cloud

- Built on an open source foundation
- Deep visibility across containers and cloud
- Radically simple to run and scale

sysdig

Creator of Falco

Practical Cloud Native Security with Falco

Risk and Threat Detection for Containers, Kubernetes, and Cloud

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Loris Degioanni and Leonardo Grasso

Practical Cloud Native Security with Falco

by Loris Degioanni and Leonardo Grasso

Copyright © 2023 Sysdig, Inc. All rights reserved.

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Jennifer Pollock

Development Editor: Sarah Grey

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2022: First Edition

Revision History for the Early Release

- 2021-11-09: First Release
- 2021-12-09: Second Release
- 2022-02-22: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098118570 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Practical Cloud Native Security with Falco*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

This work is part of a collaboration between O’Reilly and Sysdig. See our statement of editorial independence.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11857-0

Chapter 1. Introducing Falco

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Now it's time to understand Falco a little bit better. Don't worry, we'll take this easy! We will first look at what Falco does, including a high-level view on its functionality and an introductory description of each of its components. We'll follow that with the explanation of design principles that inspired Falco and still guide its development. After that, we'll explain what you can do with Falco, what is outside its domain and what you can better accomplish with other tools. Finally, we'll learn some historical context that will be useful to put things into perspective and understand the motivation that drove the development of the tool.

Falco in a Nutshell

At the highest level, Falco is pretty straightforward: you deploy it by installing multiple sensors across a distributed infrastructure. Each sensor collects data (from the local machine or by talking to some API), runs a set of rules against it and notifies you if something bad happens.

Take a look at Figure 1-1 for a high-level diagram of how it works.

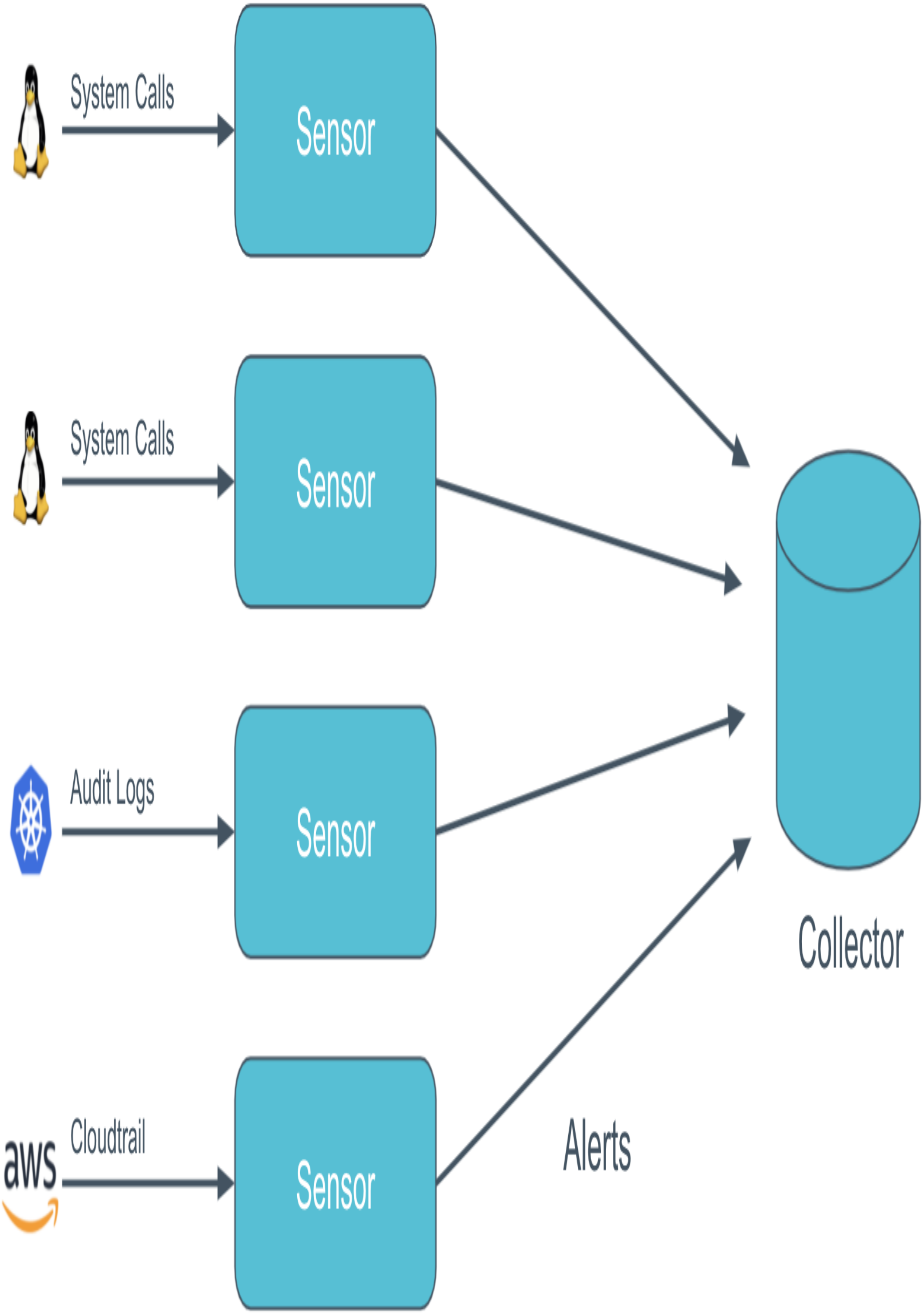

System Calls
Sensor
System Calls
Sensor
Audit Logs
Sensor
Cloudtrail
aws
Sensor
Alerts
Collector

Figure 1-1. Falco's high-level architecture

You can think about Falco as a network of security cameras for your infrastructure: you place the sensors in key locations, they observe for you what's going on and they ping you if they detect harmful behavior. With Falco, bad behavior is defined by a set of rules that the community created and maintains for you, and that you can customize for your needs.

Sensors

The architecture of a Falco sensor is shown in Figure 1-2.

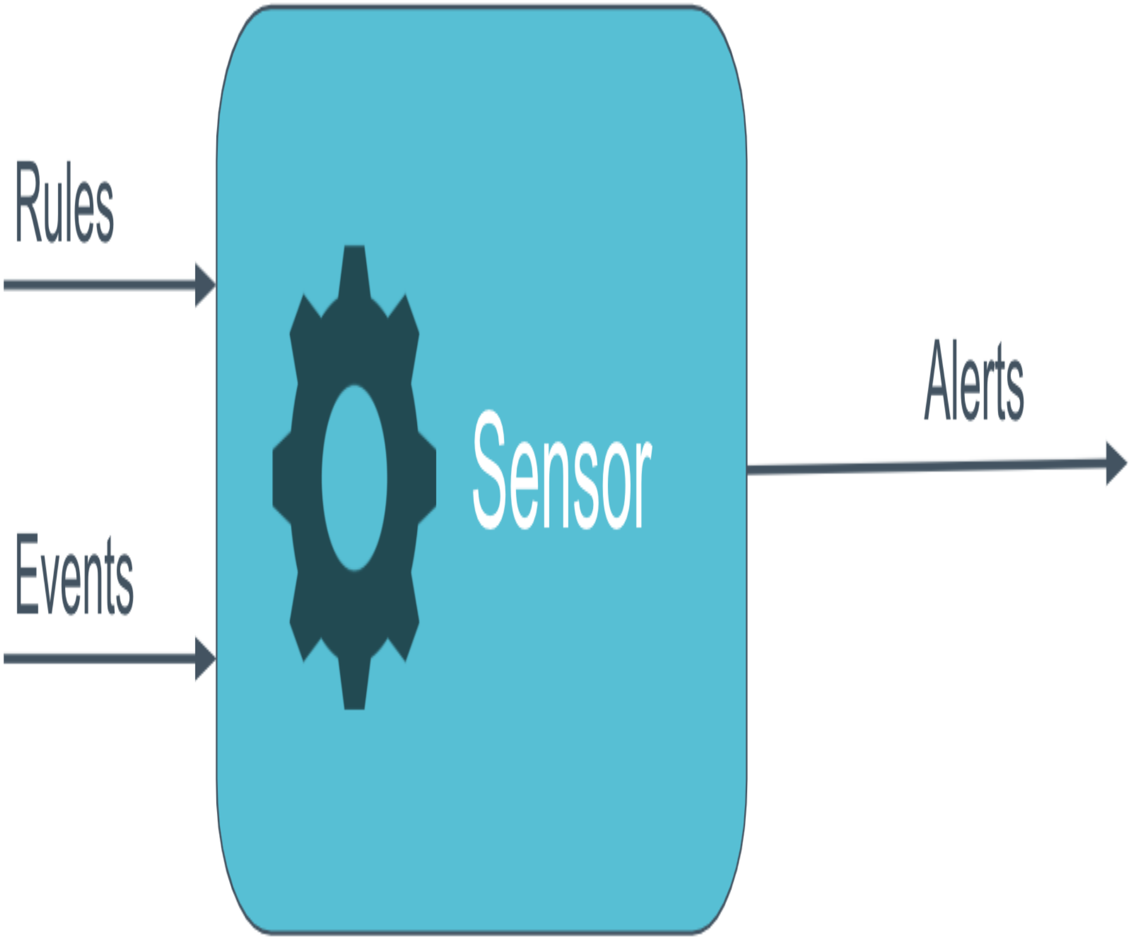


Figure 1-2. Falco sensor architecture

The sensor consists of an engine that has two inputs: a data source and a set of rules. The sensor applies the rules to each event coming from the data source. When a rule matches an event, an output message is produced. Very straightforward, right?

Data Sources

Each sensor is able to collect input data from a number of sources. Originally, Falco was designed to exclusively operate on system calls, which to date remain one of its most important data sources. We’ll cover system calls in detail in chapters 4 and 5, but for the moment, think about them as what a running program uses to interface with its external world: opening or closing a file, establishing or receiving a network connection, reading and writing data to the disk or to the network, executing commands, communicating with other processes using pipes or other types of inter process communication, these are all examples of system call usage.

Falco collects system calls by instrumenting the kernel of the Linux operating system. It can do it in two different ways: deploying a kernel module, i.e., a binary that can be installed in the operating system kernel to extend the kernel’s functionality, or using a technology called eBPF, which allows running scripts that safely perform actions inside the OS. We’ll talk extensively about kernel modules and eBPF in Chapter 5.

Tapping into this data gives Falco incredible visibility into everything that is happening in your infrastructure. Here are some examples of things Falco can detect for you:

- Privilege escalations
- Access to sensitive data
- Ownership and Mode changes
- Unexpected network connections or socket mutations
- Unwanted program execution

- Data exfiltration
- Compliance violations

But Falco is not limited to system calls. It has been extended to tap into other data sources (we'll show you examples throughout the book). For example, Falco can monitor your cloud logs in real time and notify you when something bad happens in your cloud infrastructure. Here are some more examples of things it can detect for you:

- A user logs in without multi factor authentication
- A cloud service configuration is modified
- Somebody accesses one or more sensitive files in an S3 bucket

New data sources are added to Falco frequently, so we recommend checking the website and Slack channel to keep up with what's new.

Rules

Rules tell the Falco engine what to do with the data coming from the sources. They allow the user to define policies in a compact and readable format. Falco comes pre-loaded with a comprehensive set of rules that cover host, container, Kubernetes and cloud security. However, you can easily create your own rules to customize Falco. We'll spend a lot of time on rules, in particular in Chapter 14; by the time you're done reading this book, you'll be a total master at them. For the moment, here's an example to whet your appetite:

```
- rule: shell_in_container
  desc: shell opened inside a container
  condition: container.id != host and proc.name = bash
  output: shell in a container (user=%user.name
container_id=%container.id)
  priority: WARNING
```

This rule detects when a bash shell is started inside a container, which is normally not a good thing in an immutable container-based infrastructure.

The core fields in a rule are the *condition*, which tells Falco what to look at, and the *output*, which is what Falco will tell you when the condition triggers. As you can note, both the condition and the output act on *fields*, one of the core concepts in Falco. The condition is a boolean expression that combines checks of fields against values, essentially a filter. The output is a print-like line where field values can be printed out.

Does this remind you of networking tools, like tcpdump or Wireshark? Good eye: they were a big inspiration for Falco.

Data Enrichment

Rich data sources and a flexible rule engine help make Falco a powerful runtime security tool. On top of that, metadata from a disparate set of sources enriches its detections.

When Falco tells you that something has happened, for example that a system file has been modified, you typically need more information to understand the cause and the scope of the issue. Which process did that? Did it happen in a container? If so, what were the container and image names? What was the service/namespace where this happened? Was it in production or in dev? Was this a change made by root?

Falco's data enrichment engine helps answer all of these questions by automatically attaching context to detections. It also lets you express much richer rule conditions that include this metadata: for example, you can easily scope a rule so that it only triggers in production or in a specific service. Falco accomplishes this by integrating with Kubernetes, keeping track of what runs in the cluster, and letting you use this information in rule conditions and outputs.

Outputs

Every time a rule is triggered, the corresponding engine emits an output notification. In the simplest possible configuration, the engine writes the notification on standard output (which, as you can imagine, usually isn't very useful). Fortunately, Falco offers sophisticated ways to route outputs and direct them to a bunch of places, including log collection tools, cloud storage services like S3, and communication tools like Slack and email. Its ecosystem includes a fantastic project called Falco Sidekick, specifically designed to connect Falco to the world and make output collection effortless (see Chapter 13 for more).

Containers and More

Falco was designed for the modern world of Cloud Native applications, so it has excellent, out-of-the-box support for containers, Kubernetes, and the cloud. Since this book is about Cloud Native, we will mostly focus on that, but keep in mind that Falco is not limited to containers and Kubernetes running in the cloud. You can absolutely use it as a host security tool, and many of its preloaded rules can help you secure your fleet of Linux servers. Falco has also good support for network detections, allowing you to inspect the activity of connections, IP addresses, ports, clients and servers and receive alerts when they show unwanted behavior.

Falco’s Design Principles

Now that you understand what Falco does, let’s talk about why it is the way it is.

When you’re developing a piece of software of non-negligible complexity, it’s important to focus on the right use cases and prioritize the most important goals. Sometimes that means accepting tradeoffs. Falco is no exception. Its development has been guided by a core set of principles. In this section we will learn why they were chosen and how each of them reflects on Falco’s architecture and feature set. Understanding these principles will allow you to judge if Falco is a good fit for your use cases and help you get the most out of it.

Specialized for Runtime

The Falco engine is designed to detect threats while your services and applications are running. When it detects unwanted behavior, Falco should alert you instantly (at most in a matter of seconds), so you can be informed (and react!) right away, not after minutes or hours.

This design principle manifests in three important architectural choices. First, the Falco engine is engineered as a streaming engine, able to process data quickly as it arrives rather than storing it and acting on it later. Second, it’s is designed to rely on limited state, which means correlating different events, even if feasible, is not a primary goal and is discouraged. Third, it evaluates rules as close as possible to the data source. If possible, Falco avoids transporting information before processing it and favors deploying richer engines on the endpoints.

Suitable for Production

You should be able to deploy Falco in any environment, including production environments where stability and low overhead are of paramount importance. It should not crash your apps and should strive to slow them down as little as possible.

This design principle affects the data collection architecture, particularly when Falco runs on endpoints that have many processes or containers. The Falco kernel module and eBPF probe have undergone many iterations and years of testing to guarantee their performance and stability. Collecting data by tapping into the kernel of the operating system, as opposed to instrumenting the monitored processes/containers, guarantees that your applications won’t crash because of bugs in Falco.

The Falco engine is written in C++ and employs many expedients to reduce resource consumption. For example, it avoids processing system calls that read or write disk or network data. In some ways this is a limitation, because it prevents users from creating rules that inspect the content of payloads, but it also ensures that CPU and memory consumption stay low, which is more important.

Optimized to Run at the Edge

Compared to other policy engines (for example, OPA), Falco has been explicitly designed with a distributed, multisensor architecture in mind. Its sensor is designed to be lightweight, efficient, and portable, and to operate in diverse environments. It can be deployed on a physical host, in a virtual machine, or as a container. The Falco binary is built for multiple platforms, including ARM.

Avoids Moving and Storing a Ton of Data

Most currently marketed threat-detection products are based on sending a lot of events to a centralized Security information and event management (SIEM) tool and then performing analytics on top of the collected data.

Falco is designed around a very different principle: stay as close as possible to the endpoint, perform detections in place, and only ship alerts to a centralized collector. This approach results in a solution that is a bit less capable at performing complex analytics, but is simple to operate, much more cost-effective and scales very well horizontally.

Scalable

Speaking of scale: another important design goal underlying Falco is to be able to scale to support the biggest infrastructures in the world. If you can run it, Falco should be able to secure it.

As we've just described, limited state and avoiding centralized storage are important elements of this. Edge computing, too, since distributing rule evaluation is the only approach to scale a tool like Falco in a truly horizontal way.

Another important element of scalability is endpoint instrumentation. Falco's data collection doesn't use techniques like sidecars, library linking, or process instrumentation. The reason is that the resource utilization of all of these techniques grows with the number of containers, libraries, or processes to monitor. Busy machines have many containers, libraries, and processes, too many for these techniques to work—but they have only one operating system kernel. Capturing system calls in the kernel means that you only need one Falco sensor per machine, no matter how big the machine is. This makes it possible to run Falco on big hosts with a lot of activity.

Truthful

One other benefit of using system calls as a data source? System calls never lie. Falco is hard to evade because the mechanism it uses to collect data is very hard to disable or circumvent. If you try to evade or circumvent it, you will leave traces that Falco can capture.

Sane Defaults, Richly Extensible

Another key design goal was minimizing the time it takes to extract value from Falco by just installing it; you shouldn't need to customize it unless you have advanced requirements.

Whenever the need for customization does arise, though, Falco offers flexibility. For example, you can create new rules through a rich and expressive syntax, develop and deploy new data sources that expand the scope of detections, and integrate Falco with your desired notification and event collection tools.

Simple

Simplicity is the last design choice, but it's also one of the most important ones.

The Falco rule syntax is designed to be compact, easy to read, and simple to learn. Whenever possible, a Falco rule condition should fit in a single line. Anyone, not only experts, should be able to write a new rule or modify an existing one. It is OK if this reduces the expressiveness of the syntax: Falco is in the business of delivering an efficient security rule engine, not a full-fledged domain-specific language. There are better tools for that.

Simplicity is also evident in the processes of extending Falco to alert on new data sources and of integrating it with a new cloud service or type of container, which is a matter of writing a plugin in any language, including Go, C, and C++. Falco loads these plugins easily.

What You Can Do with Falco

Falco shines at detecting threats, intrusions and data theft at runtime and in real time. It works well with legacy infrastructures but excels at supporting containers, Kubernetes and cloud infrastructures. It secures both workloads (processes, containers, services) and infrastructure (hosts, VMs, network, cloud infrastructure and services). It is designed to be lightweight, efficient and scalable and to be used in both development and production. It can detect many classes of threats, but should you need more, you can customize it. It also has a thriving community that supports it and keeps enhancing it.

What You Cannot Do with Falco

No single tool can solve all your problems. Knowing what you cannot do with Falco is as important as knowing where to use it.

As with any tool, there are tradeoffs. First of all, Falco is not a general-purpose policy language: it doesn’t offer the expressiveness of a full programming language and cannot perform correlation across different engines. Its rule engine, instead, is designed to apply relatively stateless rules at high frequency in many places around your infrastructure. If you are looking for a powerful centralized policy language, we suggest you take a look at OPA.

Second, Falco is not designed to store the data it collects in a centralized repository and let you perform analytics on it. Rule validation is performed at the endpoint and only the alerts are sent to a centralized location. If your focus is advanced analytics and big data querying, we recommend that you use one of the many log collection tools available on the market.

Finally, for efficiency reasons, Falco does not inspect network payloads. Therefore, it’s not the right tool to implement layer 7 (L7) security policies. A traditional network-based intrusion detection system (IDS) or L7 firewall is a better choice for such a use case.

Background and History

The authors of this book have been part of some of Falco’s history, and this final section brings you our memories and perspectives. If you are only interested in operationalizing Falco, feel free to skip the rest of this chapter. However, we believe that knowing where Falco comes from can give you useful context for its architecture that will ultimately help you use it better. Plus, it’s a fun story!

Network Packets: BPF, libpcap, tcpdump, and Wireshark

During the late 1990s Internet boom, computer networks were exploding in popularity. So was, the need to observe, troubleshoot, and secure them. Unfortunately, many operators couldn’t afford the network visibility tools available at that time, which were all commercially offered and very expensive. As a consequence, a lot of people were fumbling around in the dark.

Soon, teams around the world started working on solutions to this problem. Some involved extending existing operating systems to add packet-capture functionality: in other words, making it possible to convert an off-the-shelf computer workstation into a device that could sit on a network and collect all the packets sent or received by other workstations. One such solution, Berkeley Packet Filter (BPF), developed by Steven McCanne and Van Jacobson at the University of California at Berkeley, was designed to extend the BSD operating system kernel. If you use Linux, you might be familiar with eBPF, a virtual machine to safely execute arbitrary code in the Linux kernel: the “e” stands for “extended.” eBPF is one of the hottest modern features of the Linux kernel. It’s extremely powerful and flexible after many years of improvements, but it started as a little programmable packet-capture and filtering module for BSD Unix.

BPF came with a library, called libpcap, that any program could use to capture raw network packets. Its availability triggered a proliferation of networking and security tools. The first tool based on libpcap was a command-line network analyzer called tcpdump, which is still part of virtually any unix distribution. In 1998, however, a GUI-based open source protocol analyzer called Ethereal (renamed Wireshark in 2006) was launched. It became (and still is) the industry standard for packet analysis.

What tcpdump, wireshark, and many other popular networking tools have in common is the ability to access a data source that is rich, accurate, and trustworthy and can be collected in a non-invasive way: raw network packets. Keep this concept in mind as you continue reading!

Snort and Packet-Based Runtime Security

Introspection tools like tcpdump and Wireshark were the natural early applications of the BPF packet-capture stack. However, people started getting creative in their use cases for packets. For example, in 1998, Martin Roesch released an open source network-intrusion detection tool called Snort. Snort is a rule engine that processes packets captured from the network. It has a large set of rules that can detect threats and unwanted activity by looking at packets, the protocols they contain, and the payloads they carry. It inspired the creation of similar tools such as Suricata and Zeek.

What makes tools like Snort powerful is their ability to validate the security of networks and applications *while applications are running*. This is important because it provides real-time protection, and the focus on runtime behavior makes it possible to detect threats based on vulnerabilities that have not yet been disclosed.

The Network Packets Crisis

You've just seen what made network packets popular as a data source for visibility, security and troubleshooting. Applications based on them spawned several successful industries. However, the several trends eroded packets' usefulness as a source of truth:

- Collecting packets in a comprehensive way was becoming more and more complicated, especially in environments like the cloud, where access to routers and network infrastructure is limited.
- Encryption and network virtualization made it more challenging to extract valuable information.
- The rise of containers and orchestrators like Kubernetes made infrastructures more elastic. At the same time, it became more complicated to reliably collect network data.

These issues started becoming clear in the early 2010s, with the popularity of cloud computing and containers. Once again, an exciting new ecosystem was unfolding, but no one quite knew how to troubleshoot and secure it.

System Calls as a Data Source: sysdig

That's where your authors come in. We released an open source tool called sysdig. We were inspired by a set of questions: What is the best way to provide visibility for modern cloud native applications? Can we apply workflows built on top of packet capture to this new world? What is the best data source?

Sysdig originally focused on collecting system calls from the kernel of the operating system. System calls are a rich data source, even richer than packets, because they don't exclusively focus on network data: they include file I/O, command execution, interprocess communication, and more. They are a better data source for cloud native environments than packets, because they can be collected from the kernel for both containers and cloud instances. Plus, collecting them is easy, efficient, and minimally invasive.

Sysdig at that time had three separate components:

- A kernel capture probe (available in two flavors, kernel module and eBPF)
- A set of libraries to facilitate the development of capture programs
- A command-line tool with decoding and filtering capabilities

In other words, it was porting the BPF stack to system calls. Sysdig was engineered to support the most popular network-packet workflows: trace files, easy filtering, scriptability, and so on. From the beginning, we also included native integrations with Kubernetes and other orchestrators, with the goal of making them useful in modern environments. Sysdig immediately became very popular with the community, validating the technical approach.

Falco

So what would be the next logical step? You guessed it: a Snort-like tool for system calls!

A flexible rule engine on top of the sysdig libraries, we thought, would be a powerful tool to detect anomalous behavior and intrusions in modern apps reliably and efficiently. Essentially the Snort approach, but applied to system calls and designed to work in the cloud.

So, that’s how Falco was born. The first (rather simple) version was released at the end of 2016 and included most of the important components, in particular the rule engine. Falco’s rule engine was inspired by the Snort one but designed to operate on a much richer and more generic dataset and was plugged into the sysdig libraries. It shipped with a relatively small but useful set of rules. It was largely a single-machine tool, with no ability to be deployed in a distributed way. We released it as open source because we saw a broad community need for it. And, of course, because we love open source!

Expanding into Kubernetes

As the tool evolved and the community embraced it, its developers expanded it into new domains of applicability. For example, in 2018 we added Kubernetes Audit Logs as a data source. This feature lets Falco tap into the stream of events produced by the Audit Log and detect misconfigurations and threats as they happen.

Creating this feature required us to improve the engine, which made Falco more flexible and better suited to a broader range of use cases.

Joining the Cloud Native Computing Foundation

In 2018 Sysdig contributed Falco to the Cloud Native Computing Foundation (CNCF) as a sandbox project. The CNCF is the home of many important projects at the foundation of modern cloud computing, such as Kubernetes, Prometheus, Envoy, and OPA. For our team, making Falco part of the CNCF was a way to evolve it into a truly community-driven effort, to make sure it would be flawlessly integrated with the rest of the cloud native stack, and to guarantee long-term support for it. In 2021 this effort was expanded by the contribution of the sysdig kernel module, eBPF probe and libraries to the CNCF, as a subproject in the Falco organization. The full Falco stack is now in the hands of a neutral and caring community.

Plugins and the cloud

As years pass and Falco matures, a couple of things have become clear. First, its sophisticated engine, efficient nature, and ease of deployment make it suitable for much more than system-call-based runtime security. Second, as software becomes more and more distributed and complex, runtime security is paramount to immediately detecting threats, both expected and unexpected. Finally, we believe that the world needs a consistent, standardized way to approach runtime security. In particular there is great demand for a solution that can protect workloads (processes, containers, services, applications) and infrastructure (hosts, networks, cloud services) in a converged way.

As a consequence, the next step in the evolution of Falco is modularity, flexibility, and support for many more data sources spanning across different domains. In summer 2021, Falco added a new plugin infrastructure that allows it to tap into data sources like cloud provider logs to detect misconfigurations, unauthorized access, data theft, and much more.

A long journey

Falco's story stretches across more than two decades and links many people, inventions and projects that at a first glance don't look related. In our opinion, this story exemplifies why open source is so cool: becoming a contributor lets you learn from the smart people who came before you, build on top of their innovations, and connect communities in creative ways.

Chapter 2. Getting Started with Falco on Your Local Machine

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the second chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Now that you're acquainted with the possibilities that Falco offers, what could be the best way to familiarize yourself with Falco if not to try it?

In this chapter, you will discover how easy it is to install and run Falco on a local machine by following some simple instructions. We'll walk you through it step-by-step, analyzing and familiarizing you with its core concepts and functions. We will generate an event that Falco will detect for us by simulating a malicious action and learning how to read Falco's notification output. We'll finish the chapter by giving you some manageable approaches to customizing your installation.

Running Falco on Your Local Machine

Although Falco is not a typical application, installing and running it on a local machine is quite simple. All you need is a Linux host or a virtual machine and a terminal. Then you have to install two components: the userspace program (named `falco`) and a driver. The driver is needed to collect system calls, which are one possible data source for Falco. For simplicity, we will focus only on system call capture in this chapter. (We will learn more about the available drivers and why we need them to instrument the system in Chapter 3 and explore other alternative data sources in Chapter 4.) For the moment, you only need to know that the default driver, which is implemented as a Linux kernel module, is enough to collect system calls and start using Falco.

Several methods are available to install these components, as you will see in Chapter 8. However, in this chapter, we've opted for the *binary* package. It works with almost any Linux distribution and has no automation: you can touch its components with your hands. The binary package includes the `falco` program, the `falco-driver-loader` script (a utility to help us install the driver), and many other required files. You can download this package from the official website of The Falco Project (https://falco.org/), along with additional, comprehensive information about installing it. So, let's do it!

Downloading and Installing the Binary Package

The Falco's binary package is distributed as a single tarball compressed with GNU zip (gzip).

The tarball file is named `falco-`***`x.y.z-arch`***`.tar.gz`, where ***`x.y.z`*** is the version of a Falco release and ***`arch`*** is the intended architecture (e.g., `x86_64`) for the package.[1]

All packages are listed at Falco's Getting Started page. You can grab the URL of the binary package and download it locally, for example, using `curl`:

```
curl -L -O https://download.falco.org/packages/bin/x86_64/falco-
0.30.0-x86_64.tar.gz
```

After downloading the tarball, uncompressing and untarring it is quite simple:

```
tar -xvf falco-0.30.0-x86_64.tar.gz
```

The tarball content, which we have just extracted, is intended to be copied directly to the local filesystem’s root (i.e., /), without any special installation procedure. To copy it, run, as root:

```
sudo cp -R falco-0.30.0-x86_64/* /
```

The last thing we need to install is the driver.

Installing the Driver

System calls are Falco’s default data source. In order to instrument the Linux kernel and collect these system calls, it needs a driver: either a Linux Kernel Module or an eBPF probe. The driver needs to be built for the specific version and configuration of the kernel on which Falco will run. Fortunately, The Falco Project provides literally thousands of pre-built drivers for the vast majority of the most common Linux distributions, with various kernel versions available for download. If a pre-built driver for your distribution and kernel version is not yet available, the files we installed in the previous section include the source code of both the Kernel Module and the eBPF, so it is also possible to build the driver locally.

This might sound like a lot, but, the `falco-driver-loader` script you’ve just installed can do all these steps. All you need to do before using the script is install a few necessary dependencies:

- Dynamic Kernel Module Support (DKMS)
- GNU make
- The Linux kernel headers

Depending on the package manager you’re using, the actual package names can vary; however, they aren’t difficult to find.

Once you have installed the above packages, you are ready to run the `falco-driver-loader` script as a root user. If everything goes well, the script output should look something like this:

```
$ sudo falco-driver-loader
* Running falco-driver-loader for: falco version=0.30.0, driver
version=3aa7a83bf7b9e6229a3824e3fd1f4452d1e95cb4
* Running falco-driver-loader with: driver=module, compile=yes,
download=yes
* Unloading falco module, if present
* Trying to load a system falco module, if present
* Looking for a falco module locally (kernel 5.14.9-arch2-1)
* Trying to download a prebuilt falco module from
https://download.falco.org/driver/3aa7a83bf7b9e6229a3824e3fd1f445
2d1e95cb4/falco_arch_5.14.9-arch2-1_1.ko
curl: (22) The requested URL returned error: 404
Unable to find a prebuilt falco module
* Trying to dkms install falco module with GCC /usr/bin/gcc
```

From the above output, we can catch some useful information. The first line reports the versions of Falco and the driver that are being installed. The subsequent line tells us that the script will try to download a pre-built driver so it can install a Kernel Module. If the pre-built driver is not available, Falco will try to build it locally. The rest of the output shows the process of building and installing the module via DKMS, and finally, that the module has been installed and loaded.

Starting Falco

To start Falco, we just have to run it as a root user.[2]

```
$ sudo falco
Mon Oct 11 10:58:51 2021: Falco version 0.30.0 (driver version
3aa7a83bf7b9e6229a3824e3fd1f4452d1e95cb4)
Mon Oct 11 10:58:51 2021: Falco initialized with configuration
file /etc/falco/falco.yaml
Mon Oct 11 10:58:51 2021: Loading rules from file
/etc/falco/falco_rules.yaml:
```

```
Mon Oct 11 10:58:51 2021: Loading rules from file
/etc/falco/falco_rules.local.yaml:
Mon Oct 11 10:58:51 2021: Loading rules from file
/etc/falco/k8s_audit_rules.yaml:
Mon Oct 11 10:58:52 2021: Starting internal webserver, listening
on port 8765
```

Note the configuration and the rules files' paths. We'll look at these in more detail in chapters 9 and 13. Finally, in the last line, we can see that a web server has been started. Why? Falco supports various data sources, including the Kubernetes Audit Logs (see chapter 4), which delivers events through a webhook.

Once Falco prints this startup information, it is ready to issue a notification whenever a condition in the loaded ruleset is met. Right now, you probably won't see any notifications (if nothing malicious is running on your system). In the next section, we will generate a suspicious event.

Finally, keep in mind that in this chapter, to get you used to it, we have simply run Falco as an interactive shell process; a simple Ctrl + C is enough to end the process. Throughout the book, we will show you different and more sophisticated ways to install and run it.

Generating Events

There are millions of ways to generate events. In the case of system calls, in reality, many events happen continuously as soon as processes are running. However, to see Falco in action, we must focus on events that can trigger an alert. As you'll recall, Falco comes pre-loaded with an out-of-the-box set of rules that cover the most common security scenarios. In short, we must simulate a malicious action within our system!

To express unwanted behaviors, Falco uses rules. Therefore we have to pick a rule as our target. In the course of the book and particularly in Chapter 13, we will discover together the complete anatomy of a rule, how to interpret and write a condition using Falco's rule syntax, and which fields are supported in the conditions and outputs. For the moment, let us briefly recall what a rule is and explain its structure by considering a real example.

```
- rule: Write below binary dir
  desc: an attempt to write to any file below a set of binary
directories
  condition: >
    bin_dir and evt.dir = < and open_write
  output: >
    File below a known binary directory opened for writing
(user=%user.name user_loginuid=%user.loginuid
    command=%proc.cmdline file=%fd.name parent=%proc.pname
pcmdline=%proc.pcmdline gparent=%proc.aname[2]
container_id=%container.id image=%container.image.repository)
  priority: ERROR
```

A rule declaration is a YAML object with several keys. The first key, `rule`, uniquely identifies the rule within a *ruleset* (one or more YAML files containing rule definitions). The second key, `desc`, allows the rule's author to briefly describe what the rule will detect.

The `condition` key, arguably the most important one, allows expressing a security assertion using some straightforward syntax. Various boolean and comparison operators can be combined with *fields* (which hold the collected data) to filter only relevant events. Supported fields and filters are covered in more detail in Chapter 6.

As long as the condition is false, nothing will happen. The assertion is met when the condition is true, and then an alert will be fired immediately. The alert will contain an informative message, as defined by the rule's author using the `output` key of the rule. The value of the `priority` key will be reported too. The content of an alert is covered in more detail in the next section.

The `condition`'s syntax can also make use of a few more constructs, like `list` and `macro`, that can be defined in the ruleset alongside rules. As the name suggests, a *list* is a list of items that can be reused across different rules. Similarly, *macros* use pieces of conditions. For completeness, here are the two macros (bin_dir and open_write) utilized in the condition key of the *Write below binary dir* rule:

```
- macro: bin_dir
  condition: fd.directory in (/bin, /sbin, /usr/bin, /usr/sbin)
- macro: open_write
  condition: (evt.type=open or evt.type=openat) and
evt.is_open_write=true and fd.typechar='f' and fd.num>=0
```

At runtime, when rules are loaded, macros expand. Consequently, we can imagine the final rule condition will be similar to:

```
(evt.type=open or evt.type=openat) and evt.is_open_write=true and
fd.typechar='f' and fd.num>=0
and
evt.dir = <
and
fd.directory in (/bin, /sbin, /usr/bin, /usr/sbin)
```

Notably, the conditions make extensive use of fields. In the above example, you can easily recognize which parts of the condition are fields (such as evt.type, evt.is_open_write, fd.typechar, evt.dir, and fd.directory) since they are followed by a comparison operator (e.g., =, in). Fields usually contain a dot (.) in the middle, because fields with a similar context are grouped together in classes. The part before the dot represents the class (for example, evt and fd are classes).

Although you might not thoroughly understand the condition's syntax yet, you don't need to at the moment. All you need to know is that creating a file (which implies opening a file for writing) under one of the directories listed within the condition (like */bin*) should be enough to trigger the rule's condition. Let's try it.

First, start Falco with our target rule loaded. Note that the *Write below binary dir* rule is included in */etc/falco/falco_rules.yaml*, which is loaded by default when starting Falco. Luckily, you do not need to copy it manually. Just open a terminal and run:

```
sudo falco
```

Second, trigger the rule by creating a file below the */bin* directory. A straightforward way to do this is by opening another terminal and typing:

```
sudo touch /bin/surprise
```

Now, if you return to the first terminal with Falco running, you should find a line in the log (that is, an alter emitted by Falco) that looks like the following:

```
16:52:09.350818073: Error File below a known binary directory
opened for writing (user=root user_loginuid=1000 command=touch
/bin/surprise
 file=/bin/surprise parent=sudo pcmdline=sudo touch /bin/surprise
gparent=zsh container_id=host image=<NA>)
```

Falco caught us! Fortunately, that's exactly what we wanted to happen. (We'll look at this output in more detail in the next section.)

Rules let us tell Falco which security policies we want to observe (expressed by the `condition` key) and which information we wish to receive (as specified by the `output` key) if a policy has been violated. Eventually, Falco emits an alert (outputs a line of text) whenever an event meets the *condition* defined by a rule. If you run the same command again, a new alert will fire.

After trying out the example we have just given you, why not try the other rules by yourself? To facilitate this, the Falcosecurity organization offers a tool to generate events for just this reason. The `event-generator` is a simple command-line tool. It does not require special installation steps. You can download the latest release and uncompress it wherever you prefer. It comes with a collection of events that match many of the rules included in the default Falco ruleset. For example, to generate an event that meets the condition expressed by the *Change Thread Namespace* (another Falco rule), you can type the following in a terminal window:

```
$ ./event-generator run syscall.ChangeThreadNamespace
```

WARNING

Be aware that this tool might alter your system. For example, since the tool's purpose is to reproduce real malicious behavior, some actions modify files and directories below, such as /bin, /etc, and /dev. Make sure you fully understand the purpose of this tool and its options before using it.

Interpreting the Falco Output

Let's take a closer look at the alert notification in the output. What important information does it contain?

```
16:52:09.350818073: Error File below a known binary directory
opened for writing (user=root user_loginuid=1000 command=touch
/bin/surprise
 file=/bin/surprise parent=sudo pcmdline=sudo touch /bin/surprise
gparent=zsh container_id=host image=<NA>)
```

This apparently complex line is actually composed of only three main elements separated by whitespace: timestamp, severity, and message. Let's take a look at each of these.

Timestamp

Intuitively, the first element is the *timestamp* (followed by a colon: `16:52:09.350818073:`). That's when the event was generated. By default, it is displayed in the local time zone and includes nanoseconds. You can, if you like, configure Falco to display times in ISO 8601 format, including date, nanoseconds, and timezone offset (in UTC).

Severity

The second element indicates the *severity* (followed by a whitespace, e.g., `Error`) of the alert, as specified by the *priority* key in the rule. It can assume one of the following values (ordered from the most to the least severe): `Emergency`, `Alert`, `Critical`, `Error`, `Warning`, `Notice`, `Informational`, `Debug`. Moreover, it is possible to specify the minimum rule priority level we want to get in the output. So Falco allows us to filter out these alerts that are not important for us and thus reduce the noisiness. The configuration file */etc/falco/falco.yaml* allows the changing of this parameter, named `priority`, and defaulted to `Debug` (therefore, all priority levels are included by default). If, for example, we changed the configuration to `Notice` as the minimum level, then the rules with priority equal to `Informational` or `Debug` will be discarded.

Message

The last and the most essential element is the *message***.** It is a string produced according to the format specified by the `output` key. Its peculiarity lies in using placeholders, which the Falco engine replaces with the event data, as we will see in a moment.

Normally, the `output` key of a rule begins with a descriptive sentence to facilitate identifying the type of problem (e.g., `File below a known binary directory opened for writing`). Then it includes some placeholders (e.g., `%user.name`), which will be populated with actual values (e.g., `root`) when outputted. You can easily recognize placeholders since they start with a `%` symbol followed by one of the event's supported*fields*. These fields can be used in the `condition` key of a Falco rule and in the `output` key as well.

The beauty of this feature is that you can have a different output format for each security policy. This immediately gives you the most relevant information of that violation accident without having to navigate hundreds of fields.

Although this textual notification would seem to include all information and is already sufficient to be consumed by many other programs, there are other options, too. Indeed, plain text is not the only outputting format supported. If you want or if you need, Falco is also able to output notifications in JSON format by simply changing a configuration parameter. We will look at the configuration in the next section.

Using the JSON output format has the advantage of being easily parsable by consumers. When enabled, Falco will emit in output a JSON line for each alert that will look like the following, which we pretty-printed to improve readability:

```
{
  "output": "11:55:33.844042146: Error File below a known binary
directory opened for writing (user=root user_loginuid=1000
command=touch /bin/surprise file=/bin/surprise parent=sudo
pcmdline=sudo touch /bin/surprise gparent=zsh container_id=host
image=<NA>)",
  "priority": "Error",
  "rule": "Write below binary dir",
  "time": "2021-09-13T09:55:33.844042146Z",
  "output_fields": {
    "container.id": "host",
    "container.image.repository": null,
    "evt.time": 1631526933844042146,
    "fd.name": "/bin/surprise",
    "proc.aname[2]": "zsh",
    "proc.cmdline": "touch /bin/surprise",
    "proc.pcmdline": "sudo touch /bin/surprise",
    "proc.pname": "sudo",
    "user.loginuid": 1000,
    "user.name": "root"
  }
}
```

This output format reports the same text message as before. Additionally, each piece of information is separated into distinct JSON properties. You may also have noticed some extra data: for example, the rule identifier is present this time (`"rule": "Write below binary dir"`).

To try it right now, when starting Falco, simply pass a flag via the command-line argument to override the default configuration:

```
sudo falco -o json_output=true
```

Alternatively, you can edit */etc/falco/falco.yaml* and set `json_output` to `true`. This will enable the JSON format every time Falco starts, without the flag.

Customizing Your Falco Instance

When you start Falco, it loads several files. In particular, it first loads the main (and only) configuration file, as the startup log shows:

```
Thu Sep  9 14:16:03 2021: Falco initialized with configuration
file /etc/falco/falco.yaml
```

Falco looks for its configuration file at */etc/falco/falco.yaml*, by default. That's also where the provided configuration file is installed. If desired, you can specify another configuration file path using the `-c` command-line argument when running Falco. Whatever file location you prefer, the configuration must be a YAML file mainly containing a collection of key-value pairs. Let's see some available configuration options.

Rules Files

One of the most essential options, and the first you find in the provided configuration file, is the list of the rule files to be loaded:

```
rules_file:
  - /etc/falco/falco_rules.yaml
  - /etc/falco/falco_rules.local.yaml
  - /etc/falco/k8s_audit_rules.yaml
  - /etc/falco/rules.d
```

Despite the naming (for backward compatibility), `rules_file` can be either a list of rule files or a list of directories containing rule files. So, if an entry is a file, Falco reads directly. In the case of a directory, Falco will read every file in that directory.

The order matters here. The files are loaded in the presented order (files within a directory are loaded in alphabetical order). Users can customize predefined rules by simply overriding them in files that appear later in the list. For example, let’s assume that you wanted to change the *severity* of the *Write below binary dir* rule (which is included in */etc/falco/falco_rules.yaml*) from `Error` to `Emergency`. All you need to do is edit */etc/falco/falco_rules.local.yaml* (which appears later and is intended to add local overrides) and write:

```
- rule: Write below binary dir
  priority: EMERGENCY
  append: true
```

Output Channels

There is a group of options that control Falco’s available *output channels*, allowing you to specify where the security notifications should go. Furthermore, you can enable more than one simultaneously. You can easily recognize them within the configuration file (recall, you can find it in */etc/falco/falco.yaml*) since their keys are suffixed with `_output`.

By default, the only two enabled output channels are `stdout_output`, which instructs Falco to send alert messages to the standard output, and `syslog_output`, which sends them to the system logging daemon. Their configurations are:

```
stdout_output:
  enabled: true
syslog_output:
  enabled: true
```

Falco provides several other advanced built-in output channels. For example:

```
file_output:
  enabled: false
  keep_alive: false
  filename: ./events.txt
```

When `file_output` is enabled, Falco will also write its alerts to the file specified by the subkey `filename`.

Other output channels allow us to consume alerts in sophisticated ways and integrate with third parties. For instance, if you want to pass the Falco output to a local program, you can use:

```
program_output:
  enabled: false
  keep_alive: false
  program: "jq '{text: .output}' | curl -d @- -X POST
https://hooks.slack.com/services/XXX"
```

Once you enable this, Falco will execute the program for each alert and write its content to the program’s standard output. You can set any valid shell command to the `program` subkey—so this is an excellent opportunity to show off your favorite one-liners.

If you simply need to integrate with a webhook, a more convenient option is to use the `http_output`:

```
http_output:
  enabled: false
  url: http://some.url
```

A simple HTTP POST request will be sent to `url` for each alert. That makes it really easy to connect Falco to other tools, like Falcosidekick, which will forward alerts to Slack, Teams, Discord, Elasticsearch, and many other destinations.

Last but not least, Falco comes with a gRPC API and a corresponding output (such as `grpc_output`). For instance, enabling gRPC API and the gRPC output channel allows you to connect Falco to *falco-exporter*, which, in turn, will export metrics to Prometheus.

Falcosidekick and *falco-exporter* are open source projects you can find under the Falcosecurity GitHub organization. In Chapter 12, we will meet these tools again and learn how to work with outputs.

Conclusion

This chapter taught you how to install and run Falco in your local machine as a playground. You saw some simple ways to generate events and decode the output. We then looked at how to use the configuration file to customize Falco's behavior. Loading and extending rules are the primary way to instruct Falco on what to protect. Likewise, configuring the output channels empowers us to consume notifications in ways that meet our needs.

Armed with this knowledge, you can start experimenting with Falco confidently. The rest of this book will deepen what you've learned here and eventually help you master Falco completely.

[1] At the time of writing, the only officially supported architecture is x86_64. The full package list is available at https://falco.org/docs/getting-started/download/.

[2] Falco needs to run with root privilege to operate the driver that in turn collects syscalls. However, alternative approaches are possible. For example, you can learn from Falco's Running page how to run Falco in a container with the principle of least privilege.

Chapter 3. Understanding Falco’s Architecture

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the third chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Welcome to Part II of the book!

In Part I, you learned what Falco is and what it does. You also took a look at its high-level architecture, installed it on your machine, and took it for a spin. Now it’s time to step up your game!

Part II of this book (Chapters 3 through 7) will get into the inner workings of Falco. You will learn about its architecture in more detail, including its main components and how data flows across them. We’ll show you how Falco interfaces with the kernel of the operating system and with the cloud APIs to collect data, and how this data is enriched with context and metadata. Chapter 6, will also introduce you to the important topic of fields and filters. We’ll conclude Part II by talking about the outputs framework, a key piece of Falco.

In this chapter, which will familiarize you with Falco’s high-level architecture and the components of a Falco sensor. Do you really need to learn the internals of Falco in order to operate it? The answer, as often in life, is “it depends.” If your goal is to deploy Falco in its default configuration and show your boss that Falco is up and working, then you are probably fine skipping Part II of the book. If you do that, you will still be able to get things to work properly. However, some things will be hard (and some might be impossible) to do. For example, in Part II we’ll cover:

- Interpreting the Falco output
- Determining if an alert could be a false positive
- Fine-tuning Falco to privilege accuracy over noise
- Precisely adapting Falco to your environment
- Customizing and extending Falco

All of these tasks require you to truly understand the core concepts behind Falco and its architecture, and that’s what we’ll help you accomplish here.

True security is never trivial. It requires an investment that goes beyond a superficial understanding. But the investment is typically paid back in spades, because it can make the difference in whether your security is compromised and your company is in the news.

Assuming we’ve convinced you, let’s proceed by taking a look at the Falco sensor components.

Figure 3-1 depicts the main components of a typical Falco sensor deployment.

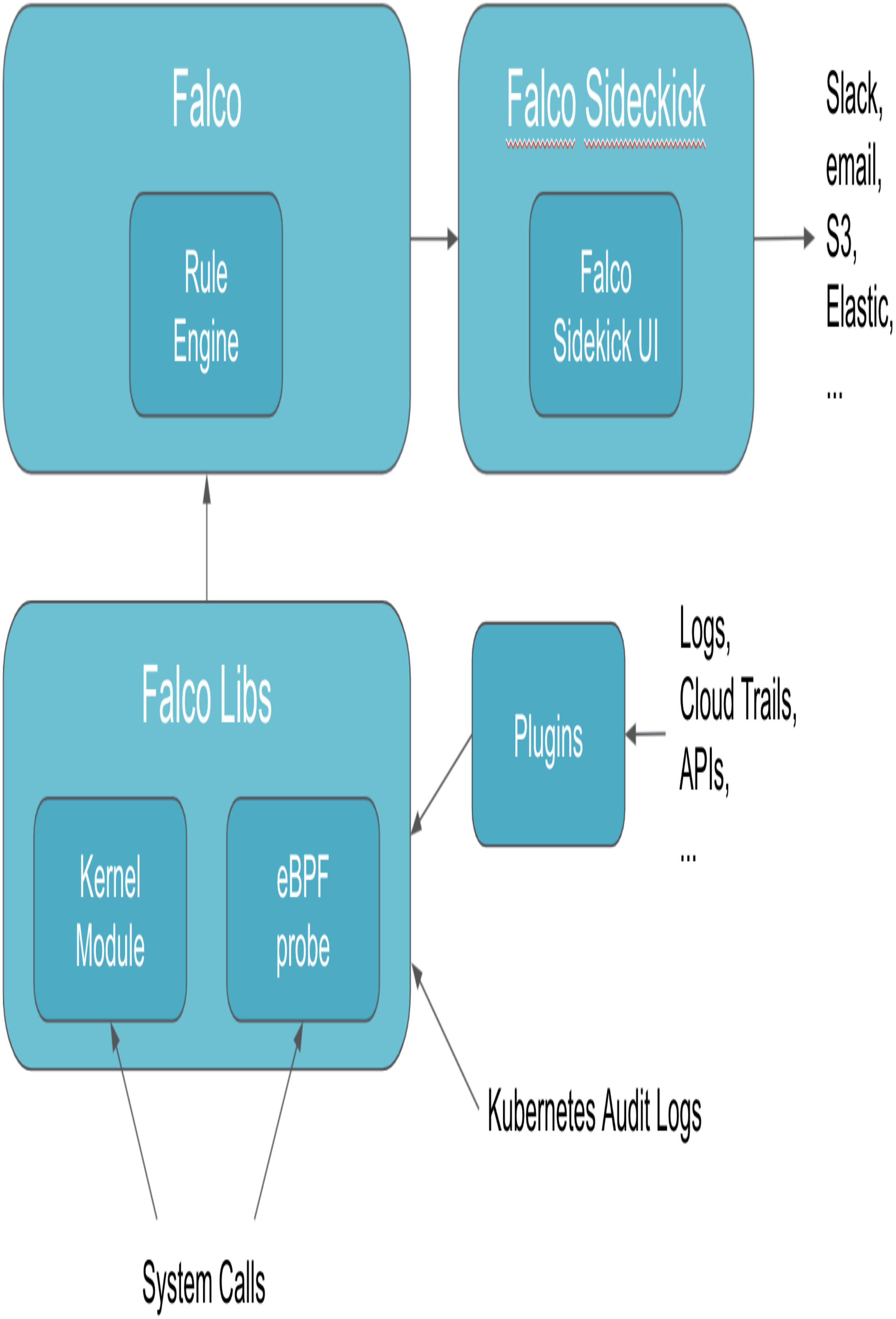

Falco
Rule Engine
Falco Sideckick
Falco Sidekick UI
Slack,
email,
S3,
Elastic,
...
Falco Libs
Kernel Module
eBPF probe
Plugins
Logs,
Cloud Trails,
APIs,
...
Kubernetes Audit Logs
System Calls

Figure 3-1. The high-level architecture of a typical Falco sensor deployment

The architecture depicted in Figure 3-1 reflects the components as they are organized at the code level in the falcosecurity organization on GitHub. At this level of granularity, the main components are:

Falco Libraries

The Falco libraries, known as “libs,” are responsible for collecting the data the sensor will process.

Plugins

The plugins extend the sensor with additional data sources.

Falco

The main sensor executable, including the rule engine

Falcosidekick

Falcosidekick is responsible for routing the notifications and connecting the sensor to the external world.

Of the components in Figure 3-1, Falco and the Falco libs are required and always installed, while Falcosidekick and the plugins are optional; you can install them based on your deployment strategy and needs.

Falco and the Falco Libraries: A Data Flow View

Let’s take the two most important of the components we just described, the Falco libs and Falco, and explore their data flow and critical modules.

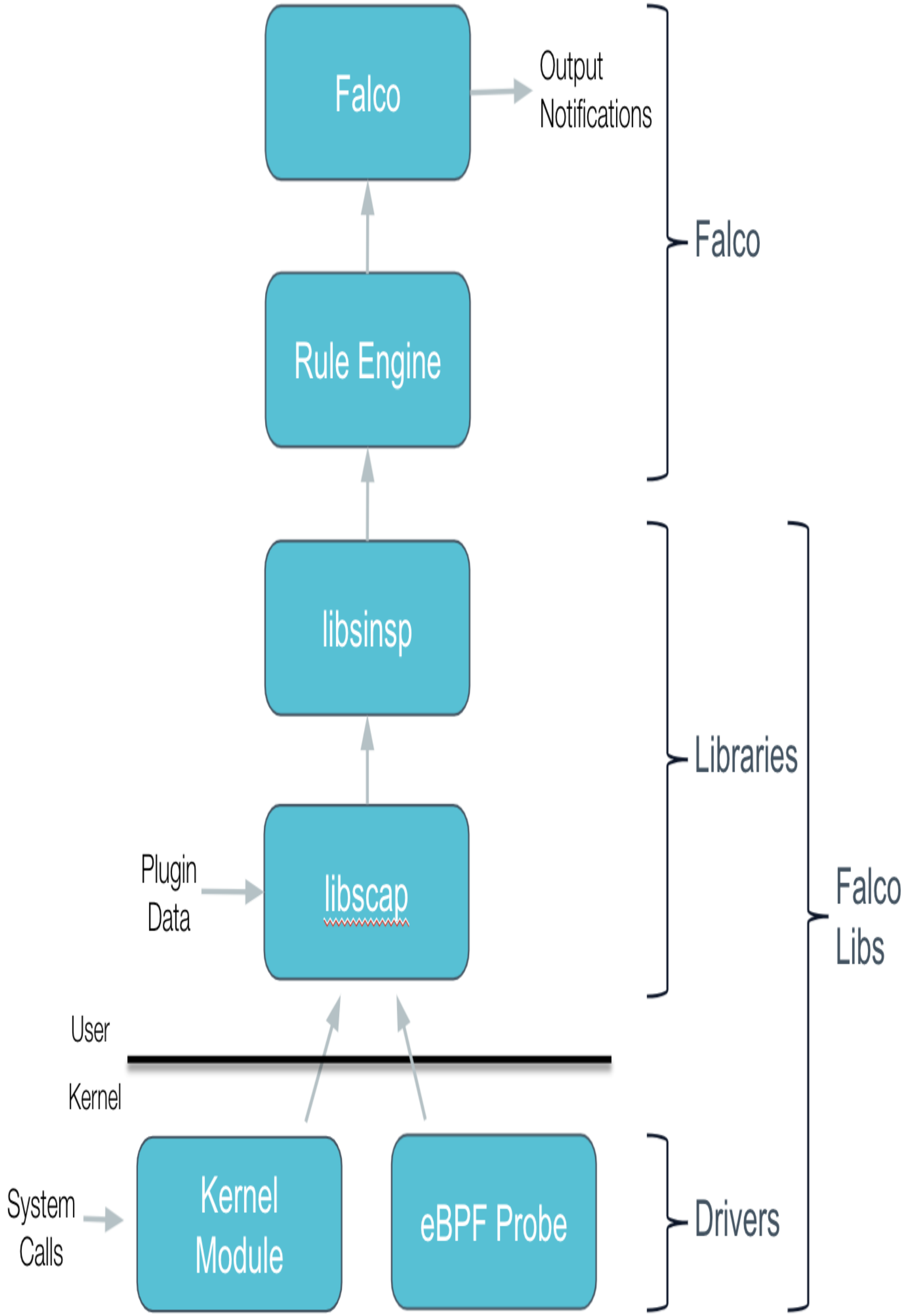

Falco
Output Notifications
Falco
Rule Engine
libsinsp
Libraries
Plugin Data
libscap
Falco Libs
User
Kernel
System Calls
Kernel Module
eBPF Probe
Drivers

Figure 3-2. Sensor data flow and main modules

You can observe, in Figure 3-2, that one of the core input sources of data is system calls. These are captured in the kernel of the operating system by one of Falco's two drivers: the *kernel module* and the *eBPF* (*extended Berkeley Packet Filter*) *probe*.

The collected system calls then flow into the first of the Falco core libraries, *libscap*, which can also receive data from the plugins and exposes a common interface to the upper layers. Data is then passed to the other key library, *libsinsp*, to be parsed and enriched. Next, the data is fed to the rule engine for evaluation. Falco receives the output of the rule engine and emits the resulting notifications, which can optionally go to Falcosidekick.

Pretty straightforward, right? Now let's dive deeper.

Figure 3-3 gives further details about what each of these modules does.

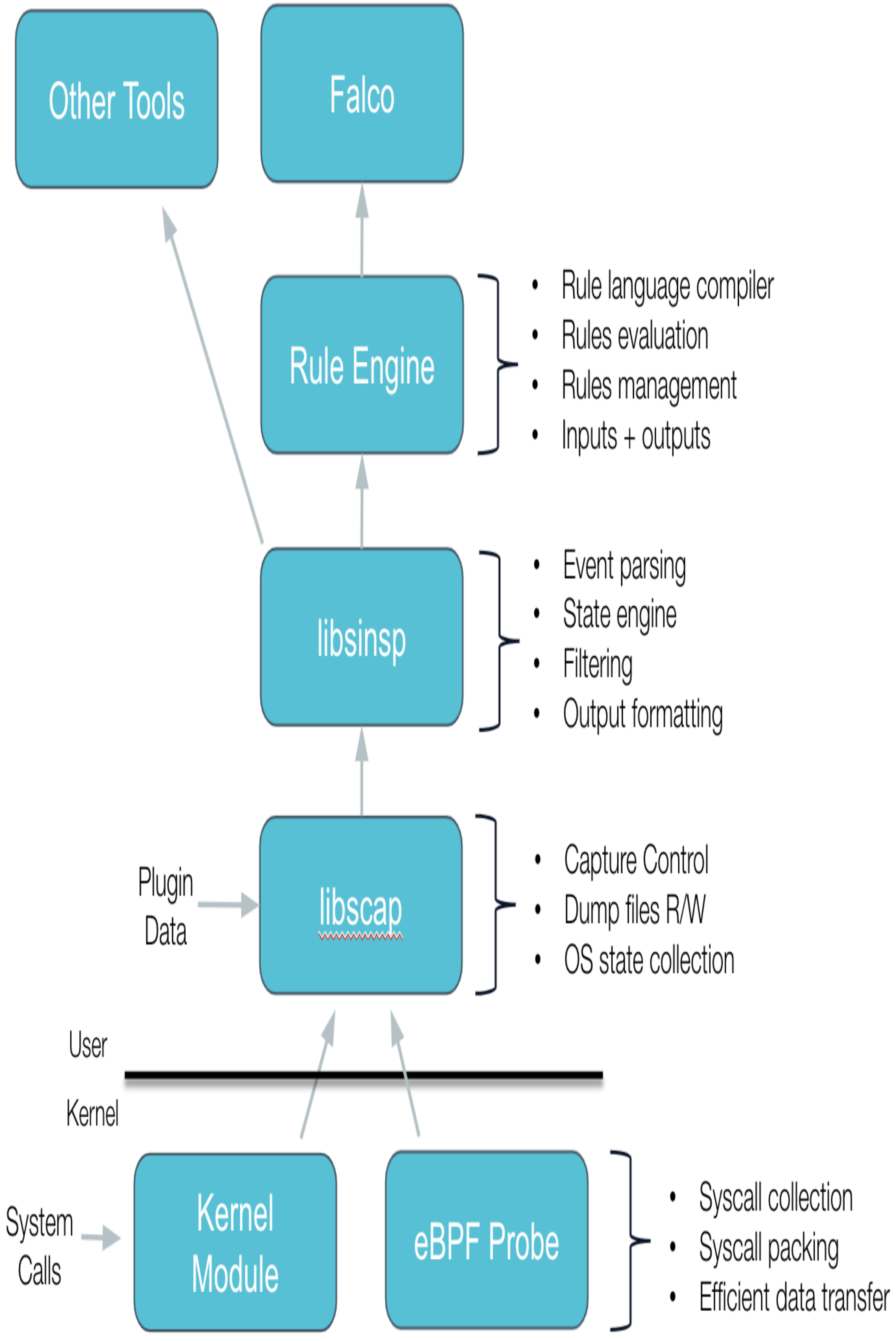

Other Tools
Falco
Rule Engine
• Rule language compiler
• Rules evaluation
• Rules management
• Inputs + outputs
libsinsp
• Event parsing
• State engine
• Filtering
• Output formatting
Plugin Data
libscap
• Capture Control
• Dump files R/W
• OS state collection
User
Kernel
System Calls
Kernel Module
eBPF Probe
• Syscall collection
• Syscall packing
• Efficient data transfer

Figure 3-3. Sensor data flow and main modules

Drivers

System calls are Falco's original data source, and to this day they remain the most important. Collecting system calls is at the core of Falco's ability to trace the behavior of processes, containers, and users in a very granular way and at high efficiency. Reliable and efficient system-call collection needs to be performed from inside the kernel of the operating system, so it requires a driver that runs inside the operating system itself. Falco offers two such drivers: the kernel module and the eBPF probe.

These two components offer identical functionality and are deployed in a mutually exclusive way: if you deploy the kernel module, you can't run the eBPF probe, and vice versa. So what is the difference, then?

The kernel module works with any version of the Linux kernel, including older ones. Also, it requires somewhat fewer resources to run, so you should use it when you care a lot about Falco having the smallest possible overhead.

The eBPF probe, on the other hand, runs only on more recent versions of Linux (starting at kernel 4.11). Its advantage is that it's safer, because its code is strictly validated by the operating system before it is executed. This means that even if it contains a bug, it is "guaranteed" not to crash your machine. Compared to the kernel module, it is also much better protected from security flaws that could compromise the machine where you run it. Therefore, in most cases, the eBPF probe is the option you should go with.

Both the kernel module and the eBPF probe are entrusted with a set of very important tasks:

Capturing system calls

> This happens through a kernel facility called tracepoints and is heavily optimized to minimize performance impact on the monitored applications.

System-call packing

Encoding system-call information into a transfer buffer, according to a format that will be easy and efficient to parse by the rest of the Falco stack.

Zero-copy data transfer

Efficiently transferring this data from the kernel to user level, where libscap will receive it. We should really call it an efficient *lack* of data transfer, since both the kernel module and the eBPF probe are designed around a *zero-copy architecture*, which maps the data buffers into user-level memory so that libscap can access the original data without needing to copy or transfer it.

In Chapter 4, you will learn all you need to know about drivers, including their architecture, functionality and usage scenarios.

Plugins

Plugins are a way to add additional data sources to Falco simply and without rebuilding Falco. Plugins implement an interface that feeds "events" into falco, similar to what the kernel module and eBPF probe do. However, plugins are not limited to capturing system calls: they can feed Falco any kind of data, including logs and API events.

Falco has several powerful plugins that extend its scope. For example, the CloudTrail plugin ingests CloudTrail JSON logs from Amazon AWS and allows Falco to alert you when something dangerous happens in your cloud infrastructure. Plugins can be written in any language, but a Go SDK makes it easier to write plugins using Go. We will talk about plugins in Chapters 4 and 14.

Library for System Capture: libscap

The term *libscap* stands for “library for system capture,” a clear hint about its purpose. libscap is the gate through which the input data goes before getting into the Falco processing pipeline. Let’s take a look at the main things libscap does for us.

Managing Data Sources

The libscap library contains the logic to control both the kernel module and the eBPF probe, including loading them, starting and stopping captures, and reading the data they produce. It also includes the logic to load, manage, and run plugins.

libscap is designed to export a generic “capture source abstraction” to the upper layers of the stack. This means that no matter what input you use to collect data (kernel module, eBPF probe, a plugin), programs that use libscap will have a consistent way to enumerate and control data sources, start andstop captures, and receive captured events, and you won’t have to worry about the nuances of interfacing with these disparate input sources.

Supporting Trace Files

Another extremely important piece of functionality in libscap is support for trace files.

If you’ve ever created or opened a pcap file with Wireshark or tcpdump, we are sure you understand how useful (and powerful!) the concept of trace files is. If not, allow us to explain.

In addition to capturing and decoding network traffic, protocol analyzers (like Wireshark and tcpdump) let you “dump” the captured network packets into a *trace file*. The trace file contains a copy of each packet, so that later you can open it to analyze the activity of that network segment. You can also share it with other people or filter its contents down to isolate relevant information.

Trace files are often referred to as *pcap files*, a name that originates from the .pcap file format used to encode the data inside them: an open, standardized format understood by every networking tool in the universe. This enables an endless list of the capture-now-analyze-later workflows that are critical in computer networks.

Many Falco users don’t realize that Falco supports trace files using the .pcap format. This feature is extremely powerful and should definitely be part of your arsenal as you become a pro.

We’ll talk extensively about how to leverage trace files, for example in Chapters 6 and 13, but for now, let’s whet your appetite by teaching you how to create a trace file and have Falco read it, in two simple steps. In order to do that, we need to introduce a command line tool called sysdig. We will learn more about sysdig in Chapter 4, but for the moment we’ll just use it as a simple trace-file generator.

Step 1: Create the trace file

Install sysdig on your Linux host by following the installation instructions. After finishing the installation, run the following on your command line, which instructs sysdig to capture all of the system calls generated by the host and write them to a file called testfile.scap.

```
$ sudo sysdig -w testfile.scap
```

Wait a few seconds to make sure your machine is working on it, then hit CTRL+c to stop sysdig.

Now you have a snapshot of a few seconds of your host’s activity. Example 3-1 shows what it contains:

Example 3-1. Sysdig test output

```
$ sysdig -r testfile.scap
1 17:41:13.628568857 0 prlcp (4358) < write res=0 data=.N;.n...
2 17:41:13.628573305 0 prlcp (4358) > write fd=6(<p>pipe:[43606])
size=1
3 17:41:13.628588359 0 prlcp (4358) < write res=1 data=.
4 17:41:13.609136030 3 gmain (2935) < poll res=0 fds=
5 17:41:13.609146818 3 gmain (2935) > write fd=4(<e>) size=8
6 17:41:13.609149203 3 gmain (2935) < write res=8 data=........
7 17:41:13.609151765 3 gmain (2935) > read fd=7(<i>) size=4096
```

```
8 17:41:13.609153301 3 gmain (2935) < read res=-11(EAGAIN) data=
9 17:41:13.626956525 0 Xorg (3214) < epoll_wait res=1
10 17:41:13.626964759 0 Xorg (3214) > setitimer
11 17:41:13.626966955 0 Xorg (3214) < setitimer
12 17:41:13.626969972 0 Xorg (3214) > recvmsg fd=42(<u>@/tmp/.X11-
unix/X0)
13 17:41:13.626976118 0 Xorg (3214) < recvmsg res=28 size=28
data=....E...................a... tuple=NULL
14 17:41:13.626992585 0 Xorg (3214) > writev fd=42(<u>@/tmp/.X11-
unix/X0) size=32
15 17:41:13.627013409 0 Xorg (3214) < writev res=32
data=...7E..........................
...
```

We'll go through the format of this output in detail later, but you can probably tell that this is a bunch of background input/output (I/O) activity performed by system tools like Xorg, gmain and prlcp, which are running on this machine while it's idle.

Step 2: Process the trace file with Falco

Think of the trace file as taking us back in time: you took a snapshot of your host at a specific point in time, and now you can trace the system calls generated on the host around that time, observing every process in detail. Processing the trace file with Falco is easy and lets you see quickly if any security violations happened during that time. Example 3-2 shows a sample of its output.

Example 3-2. Processing a trace file with Falco

```
$ ./falco -c ../../../falco.yaml -r ../../../rules/falco_rules.yaml
-e testfile.scap
Wed Sep 29 18:04:00 2021: Falco version 0.30.0
Wed Sep 29 18:04:00 2021: Falco initialized with configuration file
../../../falco.yaml
Wed Sep 29 18:04:00 2021: Loading rules from file
../../../rules/falco_rules.yaml:
Wed Sep 29 18:04:00 2021: Reading system call events from file:
testfile.scap
Events detected: 0
Rule counts by severity:
Triggered rules by rule name:
Syscall event drop monitoring:
   - event drop detected: 0 occurrences
   - num times actions taken: 0
```

Fortunately, it looks like we're safe. This consistent, back-in-time way of running Falco is useful when writing or unit-testing rules. We'll talk more about it when we deep dive into rules in chapter 13.

System State Collection

System state collection is an important task that's specifically related to capturing system calls. The kernel module and the eBPF probe produce "raw" system calls, which lack some important context Falco needs.

Let us show you an example. A very common system call is `read`, which, as the name implies, reads a buffer of data from a file descriptor. Here is the prototype of `read`:

```
ssize_t read(int fd, void *buf, size_t count);
```

It has three inputs: the numeric file descriptor identifier, a buffer to fill, and the buffer size. It returns the amount of data was written in the buffer.

A *file descriptor* is like the ID of an object inside the operating system kernel: it can indicate a file, a network connection (specifically a socket), the endpoint of a pipe, a *mutex* (used for process synchronization), a timer, or several other types of objects.

Knowing the file descriptor number is not very useful when crafting a Falco rule. As users, we prefer to think about a file or directory name, or maybe a connection's IP addresses and ports, than a file descriptor number. libscap helps us do that. When Falco starts, libscap fetches a bunch of data from a diverse set of sources within the operating system, for example the /proc Linux file system. It uses this data to construct a set of tables that can be used to resolve cryptic numbers (file descriptors, process IDs, and so forth) into logical entities and their details, which are much easier for humans to use.

This functionality is part of why Falco's syntax is much more expressive and usable than most comparable tools. One theme that you will be hearing often in this book is: *Granular data is useless without context*. Now you can start understanding what we mean by that. To continue on this topic, let’s talk about the other important Falco library.

libsinsp

The term *libsinsp* stands for “library for system inspection.” Libsinsp taps into the stream of data libscap produces, enriches it, and offers a number of higher-level primitives to work with it. Let’s take a look at its most important functionality: the state engine.

State Engine

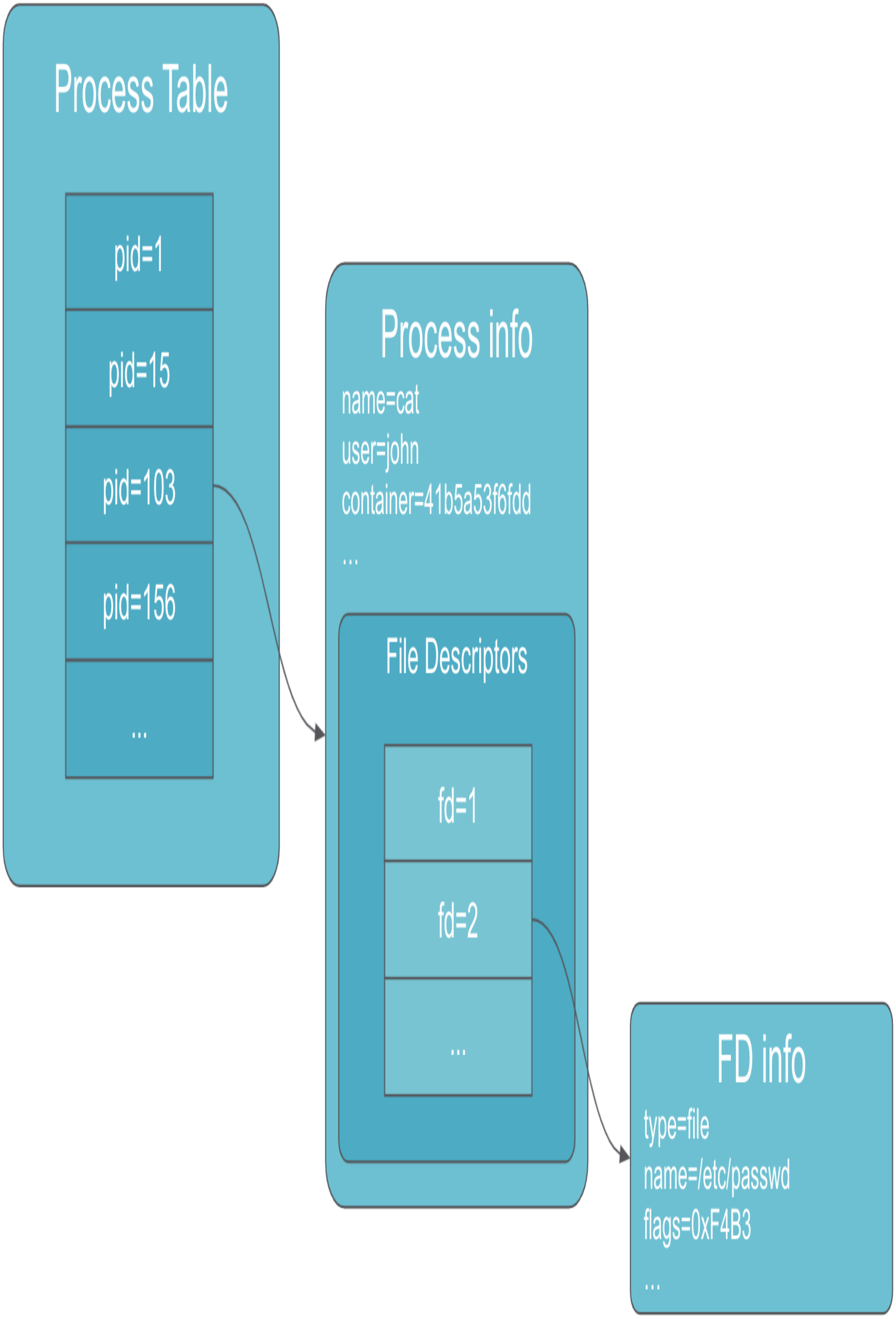
As we noted earlier, when Falco starts, libscap constructs a set of tables to convert low-level identifiers, like file descriptor numbers, into high-level, actionable information, like IP addresses and file names. This is great, but what if a program opens a file *after* Falco starts? For example, a very common system call in unix is `open`, which takes two input arguments, the file name, and some flags and returns a file descriptor identifying the newly opened file:

```
int open(const char *pathname, int flags);
```

In practice, `open`, like many other system calls, creates a new file descriptor, effectively changing the state of the process that called it. If a process invokes `open` after Falco has been launched, its new file descriptor will not be part of the state and Falco won’t know what to do with that descriptor. However, consider this: `open` is a system call. More generally, system calls are always used to createg, destroy, or modify file descriptors. Recall, too, that the Falco libs capture *all* system calls from *every* process.

Libsinsp, in particular, has logic to inspect every state-changing system call and, based on the system-call arguments, update the state table. In other words, it tracks the activity of the whole machine to keep the state always in sync with the underlying operating system. Further, it does so in a way that accurately supports containers.

Libsinsp keeps this constantly updated information in a hierarchical structure. This structure (Figure 3-4) starts from a process table, each entry of which contains a file descriptor table, among other information.



Process Table
pid=1
pid=15
pid=103
pid=156
...
Process info
name=cat
user=john
container=41b5a53f6fdd
...
File Descriptors
fd=1
fd=2
...
FD info
type=file
name=/etc/passwd
flags=0xF4B3
...

Figure 3-4. The libsinsp state hierarchy

This accurate, constantly updated state table is the core of Falco's data enrichment, which in turn is a key building block of the rule engine.

Event Parsing

The state engine requires a substantial amount of logic to understand system calls and parse their arguments. This is what libsinsp's *event parser* does.

State tracking leverages event parsing, but it's used for other purposes as well. For example, it extracts useful arguments from system calls or other data sources, making them available to the rule engine. It also collates and reconstructs buffers that can be spread across multiple collected messages, making it easier to decode their content from Falco rules.

Filtering

Filtering is one of the most important concepts in Falco, and it's fully implemented in libsinsp. A *filter* is a boolean expression that ties together multiple *checks*, each of which compares a *filter field* with a constant value. The importance of filters is obvious when we look at rules. (Indeed, it's so important that we dedicate all of chapter 6 to it.) Let's take the simple rule shown in Example 3-3.

Example 3-3. A simple Falco rule

```
- rule: shell_in_container
  desc: shell opened inside a container
  condition: container.id != host and proc.name = bash
  output: shell in a container (user=%user.name
container_id=%container.id)
  priority: WARNING
```

The `condition` section of the rule is a libsinsp filter. The condition in our example checks that the container id is not `host` and that the name of the process is `bash`. Every captured system call that meets both criteria will trigger the rule.

Libsinsp is responsible for defining and implementing system-call-related filter fields. It also contains the engine that evaluates filters and tells us if the rule should trigger, so it’s not an exaggeration to say that libsinsp is the heart of Falco!

Output Formatting

By looking at Example 3-3 again, we can see that another section, *output*, makes use of a syntax that reminds a bit of the one of the *condition* section.

```
  output: shell in a container (user=%user.name
container_id=%container.id)
```

Output is what Falco prints when the rule triggers—and yes, you can use filter fields in it, the same fields that you can use in the condition, by prepending the *%* character to the field names. libsinsp has the logic to resolve these fields and create the final output string. What’s nice is that if you become an expert at writing condition filters, you also will have mastered output strings!

One More Thing About libsinsp

By now you can probably see that a lot of Falco’s logic is in libsinsp. That’s deliberate. Falco’s developers recognized the value (and elegance) of its data-collection stack and realized it could be the base for many other tools. That’s precisely why libsinsp exists. It sits on top of the powerful Falco collection stack (which includes the drivers, plugins, and libscap) and adds the most important pieces of the Falco logic in a way that makes them reusable. What’s more, libsinsp includes all you need to collect security and forensics data from containers, virtual machines, Linux hosts, and cloud infrastructure. It’s stable, efficient, and is well documented.

Several other open source and commercial tools have been built on top of libsinsp. If you would like to write one, or if you are just curious to learn more, we recommend you start at the Libs repository.

Rule Engine

The Falco rule engine is the component you interact with when you run Falco. Here are some of the things that the rule engine is responsible for:

- Loading Falco rule files
- Parsing the rules in a file
- Using libsinsp to compile the condition and output of each rule
- Performing the appropriate action, including emitting the output, when a rule triggers

Thanks to the power of libscap and libsinsp, the rule engine is simple and relatively independent from the rest of the stack.

Conclusion

There you have it. Now you know what's inside Falco and how its components relate to each other. Your mastery of Falco is well under way.

In the next chapters we'll dive deeper into some of the components and concepts that this chapter describes.

Chapter 4. Data Sources

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

In this chapter we go deep—deep into the kernel of the operating system and into Falco's data collection stack. You'll learn how Falco captures the different types of events that feed its rule engine. You'll see how its data collection compares to alternative approaches and why it was built the way it is. You'll get to understand the details well enough that you will be able to pick and deploy the right drivers and plugins by the end of this chapter.

The first order of business is understanding what data sources you can use in Falco. Falco's data sources can be grouped into two main families: system calls and plugins.

System calls are Falco's original data source. They come from the kernel of the operating system and offer visibility into the activities of processes, containers, virtual machines and hosts. Falco uses them to protect workloads and applications.

Our second family of data sources, *plugins,* is relatively recent: it was added in 2021. It connects a varied number of inputs to Falco, such as cloud logs.

There is a third, separate source type in the works: the Kubernetes audit log, which the Falco team plans to implement as a plugin in the future. Since it is not yet available, we will not focus our attention on it, but you can take a look at the documentation page. Let's look at the first two families in turn.

System Calls

As we have repeated many times, system calls are a key source of data for Falco and one of the ingredients that make it unique. But what exactly is a system call? Let's start with a high-level definition, courtesy of Wikipedia:

> *In computing, a system call (commonly abbreviated to syscall) is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed. This may include hardware-related services (for example, accessing a hard disk drive or accessing the device's camera), creation and execution of new processes, and communication with integral kernel services such as process scheduling.*

Let's unpack this.

At the highest level of abstraction, a computer consists of a bunch of hardware that runs a bunch of software. In modern computing, however, it's extremely unusual for a program to run directly on the hardware. Instead, the vast majority of cases programs run on top of an operating system. Falco's drivers focus specifically on the operating system powering the cloud and the modern data center: Linux.

An *operating system* is a piece of software designed to conduct and support the execution of other software. Among many other things, the operating system takes care of:

- Scheduling processes
- Managing memory
- Mediating hardware access

- Implementing network connectivity
- Handling concurrency

Clearly, the vast majority of this functionality needs to be exposed to the programs that are running on top of the operating system, so that they can do something useful. And, clearly, the best way for a piece of software to expose functionality is to offer an *application programming interface*, better known as an *API*: a set of functions that client programs can call. This is what system calls *almost* are: an API of functions to interact with the operating system.

Wait, why *almost*?

Well, the operating system is a unique piece of software, so you cannot just *call* it like you would a library. The operating system runs in a separate execution mode, called *privileged mode*, which is isolated from *user mode*, which is the context of executing regular processes (that is, running programs). . This separation makes calling the OS more complicated. With some CPUs, you invoke a system call by triggering an interrupt. With most modern CPUs, however, you need to use a specific CPU instruction. If we exclude this additional level of complexity, however, it is fair to say that system calls are APIs to access operating-system functionality. There are lots of them, each with input arguments and a return value.

Every program, with no exceptions, makes extensive and constant use of the system call interface for anything that is not pure computation: reading input, generating output, accessing the disk, communicating on the network, running a new program, and so on. This means, as you can imagine, that observing system calls gives a very detailed picture of what each process does.

Operating system developers have long treated the system call interface as a stable API. This means that you can expect it to stay the same even if, inside, the kernel changes dramatically. This is important because it guarantees consistency across time and execution environments, making the system call API an even better choice for collecting reliable security signals. Falco rules, for example, can reference specific system calls and assume that using them will work on any Linux distribution.

Examples

Linux offers *many* system calls: over 300 of them. Going over all of them would be impossible and very boring, so we will spare you from it. However, we want to give you a flavor of what system calls are available.

Table 4-1 includes some of the system call categories that are most relevant for a security tool like Falco. For each category, the table includes examples of representative system calls. You can find more information on each by running `man 2 X`, where X is the system call name, in a Linux terminal or in the google search bar.

Table 4-1. Noteworthy system callc

ategories

Category	Examples
File I/O	open, creat, close, read, write, ioctl, link, unlink, chdir, chmod, stat, seek, mount, rename, mkdir, rmdir
Network	socket, bind, connect, listen, accept, sendto, recvfrom, getsockopt, setsockopt, shutdown
Inter process communication	pipe, futex, inotify_add_watch, eventfd, semop, semget, semctl, msgctl
Process management	clone, execve, fork, nice, kill, prctl, exit, setrlimit, setpriority, capset
Memory management	brk, mmap, mprotect, mlock, madvise
User management	setuid, getuid, setgid, getgid
System	sethostname, setdomainname, reboot, syslog, uname, swapoff, init_module, delete_module

TIP

If you are interested in taking a look at the full list of Linux system calls, type `man syscalls` into a Linux terminal or a search engine. This will show the official Linux manual page, which includes a comprehensive list of system calls, with hyperlinks to take a deeper look at many of them. In addition, software engineer Filippo Valsorda offers on his personal homepage a nicely organized and searchable list.

Observing System Calls

Given how crucial system calls are for Falco and for runtime security in general, it is important that you learn how to capture, observe, and interpret them. This is a valuable skill that you will find useful in many situations. We want to show you two different tools you can use for the purpose: strace and sysdig.

strace

strace is a tool that you can expect to find on pretty much every machine running a Unix-compatible operating system. In its simplest form, you use it to run a program, and it will print every system call issued by the program on standard error. In other words: add *strace* to the beginning of an arbitrary command line and you will see all of the system calls that command line generates:

```
$ strace echo hello world
execve("/bin/echo", ["echo", "hello", "world"], 0x7ffc87eed490 /*
32 vars */) = 0
brk(NULL)                               = 0x558ba22bf000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file
or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file
or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=121726, ...}) = 0
mmap(NULL, 121726, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f289009c000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file
or directory)
```

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0\0
"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030928, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f289009a000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f288faa0000
mprotect(0x7f288fc87000, 2097152, PROT_NONE) = 0
mmap(0x7f288fe87000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) =
0x7f288fe87000
mmap(0x7f288fe8d000, 15072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f288fe8d000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f289009b540) = 0
mprotect(0x7f288fe87000, 16384, PROT_READ) = 0
mprotect(0x558ba2028000, 4096, PROT_READ) = 0
mprotect(0x7f28900ba000, 4096, PROT_READ) = 0
munmap(0x7f289009c000, 121726)          = 0
brk(NULL)                               = 0x558ba22bf000
brk(0x558ba22e0000)                     = 0x558ba22e0000
openat(AT_FDCWD, "/usr/lib/locale/locale-archive",
O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3004224, ...}) = 0
mmap(NULL, 3004224, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7f288f7c2000
close(3)                                = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) =
0
write(1, "hello world\n", 12hello world (1)
)           = 12
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Note how strace's output mimics the C syntax and looks like a stream of function invocations, with the addition of the return value after the = symbol at the end of each line. For example, at (1) we can see the *write* syscall that outputs the "hello world" string to standard output (file descriptor 1). It returns the value 12, which is the number of bytes that have been successfully written. Note how the string "hello world" is printed on standard output *before* the call to the write system call returns and strace prints its return value on the screen.

A second way to use strace is pointing it to a running process by specifying the process ID (PID) on the command line:

```
$ sudo strace -p`pidof vi`
strace: Process 16472 attached
select(1, [0], [], [0], NULL)           = 1 (in [0])
read(0, "\r", 250)                      = 1
select(1, [0], [], [0], {tv_sec=0, tv_usec=0}) = 0 (Timeout)
select(1, [0], [], [0], {tv_sec=0, tv_usec=0}) = 0 (Timeout)
write(1, "\7", 1)                       = 1
select(1, [0], [], [0], {tv_sec=4, tv_usec=0}) = 0 (Timeout)
select(1, [0], [], [0], NULL
^C
strace: Process 16472 detached
<detached ...>
```

strace has some pros and some cons. It is broadly supported, so either it’s already available or it’s an easy package install away. It is simple to use and ideal when you need to inspect a single process, which makes it perfect for debugging use cases.

As for disadvantages, strace instruments individual processes, which makes it unsuitable for inspecting the activity of the whole system, or when you don’t have a specific process to start from. Further, strace is based on ptrace for system call collection, which makes it very slow and unsuitable for use in production environments. You should expect a process to slow down substantially (sometimes by orders of magnitude) when you attach strace to it.

sysdig

We introduced sysdig with our discussion of trace files in Chapter 3. Sysdig is more sophisticated than strace and includes some advanced features. While this can make it a bit harder to use, the good news is that sysdig shares Falco’s data model, output format, and filtering syntax—so you can use a lot of what you learn about Falco in sysdig, and vice versa.

The first thing to keep in mind is that you don’t point sysdig to an individual process like you do with strace. Instead, you just run it and it will capture every system call invoked on the machine, inside or outside containers:

```
$ sudo sysdig
1 17:41:13.628568857 0 prlcp (4358) < write res=0 data=.N;.n...
2 17:41:13.628573305 0 prlcp (4358) > write fd=6(<p>pipe:[43606])
size=1
4 17:41:13.609136030 3 gmain (2935) < poll res=0 fds=
5 17:41:13.609146818 3 gmain (2935) > write fd=4(<e>) size=8
6 17:41:13.609149203 3 gmain (2935) < write res=8 data=........
9 17:41:13.626956525 0 Xorg (3214) < epoll_wait res=1
10 17:41:13.626964759 0 Xorg (3214) > setitimer
11 17:41:13.626966955 0 Xorg (3214) < setitimer
```

Usually, this is too noisy and not very useful.

The way you restrict what sysdig shows you is by using filters. Sysdig accepts the same filtering syntax as Falco (which, incidentally, makes it a great tool to test and troubleshoot Falco rules). Here’s an example where we restrict sysdig to capturing system calls for processes named “echo”:

```
$ sudo sysdig proc.name=echo & /bin/echo hello world
[5] 2118
hello world
1562403 13:26:21.845954342 1 echo (2119) < execve res=0
exe=/bin/echo args=hello.world. tid=2119(echo) pid=2119(echo)
ptid=5184(zsh) cwd= fdlimit=1024 pgft_maj=1 pgft_min=55
vm_size=364 vm_rss=4 vm_swap=0 comm=echo
cgroups=cpuset=/.cpu=/user.slice.cpuacct=/user.slice.io=/user.sli
ce.memory=/user.slic... env=HOME... tty=34817 pgid=2119(echo)
loginuid=-1
1562404 13:26:21.846003757 1 echo (2119) > brk addr=0
1562405 13:26:21.846005566 1 echo (2119) < brk res=559C1160A000
vm_size=364 vm_rss=4 vm_swap=0
1562407 13:26:21.846123016 1 echo (2119) > access mode=0(F_OK)
1747052 13:26:21.845954342 1 echo (2119) < execve res=0
exe=/bin/echo args=hello.world. tid=2119(echo) pid=2119(echo)
ptid=5184(zsh) cwd= fdlimit=1024 pgft_maj=1 pgft_min=55
vm_size=364 vm_rss=4 vm_swap=0 comm=echo
cgroups=cpuset=/.cpu=/user.slice.cpuacct=/user.slice.io=/user.sli
ce.memory=/user.slic... env=HOME... tty=34817 pgid=2119(echo)
```

```
loginuid=-1
1562408 13:26:21.846133680 1 echo (2119) < access res=-2(ENOENT)
name=/etc/ld.so.nohwcap
1747053 13:26:21.846003757 1 echo (2119) > brk addr=0
```

This output requires a little bit more explanation than the strace one. The fields sysdig prints here are:

- Incremental event number
- Event timestamp
- CPU ID
- Command name
- Thread ID (tid)
- Event direction. “>” means *enter*, while “<” means *exit*
- Event type. For our purpose, this is the system call name
- System call arguments

Unlike strace, sysdig prints *two* lines for each system cal. One is called *enter* and is generated when the system call starts. The other one is called *exit* and identifies when the system call returns. This approach works well if you need to identify how long a system call took to run or pinpoint a process that is stuck in a system call.

The second thing to note is that, by default, sysdig prints thread IDs, not process IDs. *Threads* are the core execution unit for the operating system and thus for sysdig as well. Multiple threads can exist within the same process or command and share resources, such as memory. The *thread ID (*TID) is the basic identifier to follow when tracking execution activity in your machine. You do that by just looking at the TID number, or by filtering out the noise with a command line like this one:

```
sysdig thread.tid=1234
```

The example in this snippet will only preserve the execution flow for thread 1234.

Threads live inside processes, which are identified by a *process ID* (PID). Most of the processes running on an average Linux box are single-threaded, and in that case `thread.tid` is the same as `proc.pid`. Filtering by `proc.pid` is useful to observe how threads interact with each other inside a process.

Trace files

As you learned in Chapter 3, you can instruct sysdig to save the system calls it captures to a trace file:

```
$ sudo sysdig -w testfile.scap
```

You will likely want to use a filter to keep the file size under control:

```
$ sudo sysdig -w testfile.scap proc.name=echo
```

You can also use filters when reading trace files:

```
$ sysdig -r testfile.scap proc.name=echo
```

Sysdig's filters are important enough that we will devote a full chapter (Chapter 7) to them.

We recommend you play with sysdig and explore the activity of common programs in Linux. This will help build the muscle that you will use when creating or interpreting Falco rules.

Capturing System Calls

All right, system calls are cool and we need to capture them. So what is the best way to do it?

Earlier in this chapter, we described how system calls involve transitioning the execution flow from a running process to the kernel of the operating system. Intuitively, and as shown in Figure 4-1, there are two places where system calls can be captured: the running process and the operating system.

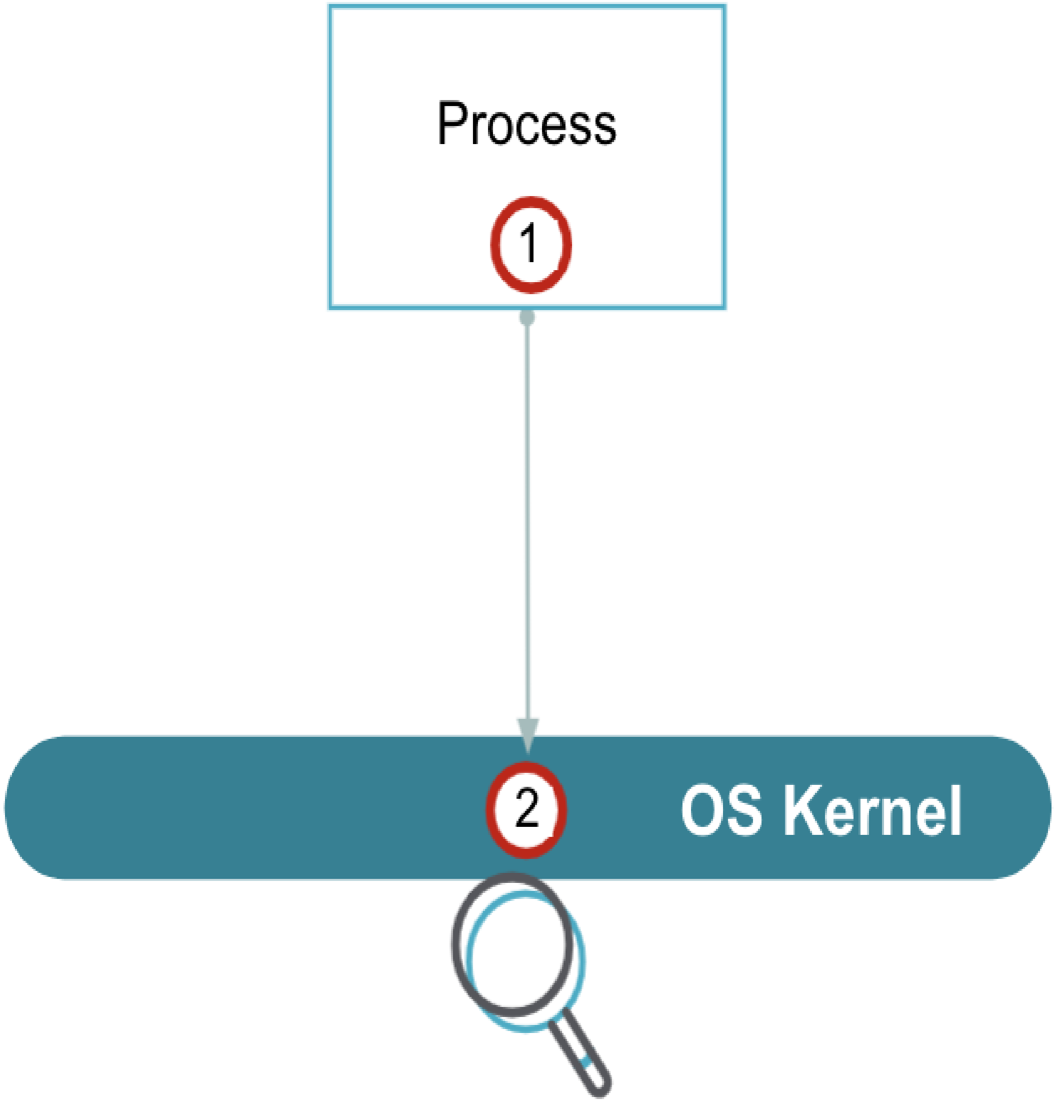


Figure 4-1. System call capture options

Capturing in the running process typically involves modifying either the process or some of its libraries with some kind of instrumentation.

The fact that most programs in Linux use the C standard library, also known as *glibc*, to execute system calls makes instrumenting it quite appealing. As a consequence, there are abundant tools and frameworks to modify glibc and other system libraries for instrumentation purposes. These techniques can be static (they change the source code of glibc and recompile it) or dynamic (they find the location of glibc in the address space of a target process and insert hooks in it).

Another method to capture system calls without instrumenting the OS kernel consists in using the operating system's debugging facilities. For example, strace uses a facility called *ptrace,*[1] which is at the base of tools like gdb.

Option number 2 in Figure 4-1, capturing in the kernel, involves intercepting the system call execution after it has transitioned to the operating system. This requires running some code in the kernel itself. It tends to be more delicate and, potentially, riskier, because running code in the kernel requires elevated privileges. Anything running in the kernel has potential control of the machine, its processes, its users, and its hardware. Therefore, a bug in anything that runs inside the kernel can cause major security risks, data corruption, or, in some cases, even a machine crash. This is why many security tools pick instrumentation option 1 and capture system calls at the user level, inside the process.

Falco does the opposite: it sits squarely on the kernel instrumentation side. The rationale behind this choice can be summarized in three words: accuracy, performance, and scalability. Let us explain each in turn.

Accuracy

User-level instrumentation techniques, in particular those that work at the glibc level, have a couple of major problems.

First, a motivated attacker can evade them by, well, not using glibc! You don't *have* to use a library to issue system calls, and attackers can easily craft a simple sequence of CPU instructions instead, completely bypassing the glibc instrumentation. Not good.

Even worse, major categories of software just don’t load glibc at all. For example, statically linked C programs, very common in containers, import glibc functions at compile time and embed them in their executable. With these programs, there’s no glibc to replace or modify. Another example is the Go language, which has its own statically linked system-call interface library.

Kernel-level capture doesn’t suffer from these limitations. It supports any language, any stack, and any framework, because system call collection happens below all of the libraries and abstraction layers. This means that kernel-level instrumentation is much harder for attackers to evade.

Performance

Some user-level capture techniques, ptrace for example, have significant overhead because they generate a high number of context switches. In other words, every single system call needs to be uniquely delivered to a separate process, which requires the execution to ping-pong between processes. This is very, very slow, to the point that it becomes an impediment to using such techniques in production, where such a substantial impact on the instrumented processes is not acceptable.

It’s true that glibc-based capture can be more efficient, but it still introduces high overhead for basic operations like timestamping events. Kernel-level capture, by contrast, requires zero context switches and can collect all of the necessary context, like timestamps, from within the kernel. This makes it much faster than any other technique, and thus the most suitable for production.

Scalability

As the name implies, process-level capture requires “doing something” for every single process. What this something is can vary, but it still introduces an overhead that is proportional to the number of observed processes. That doesn’t happen with kernel-level instrumentation. Take a look at Figure 4-2.

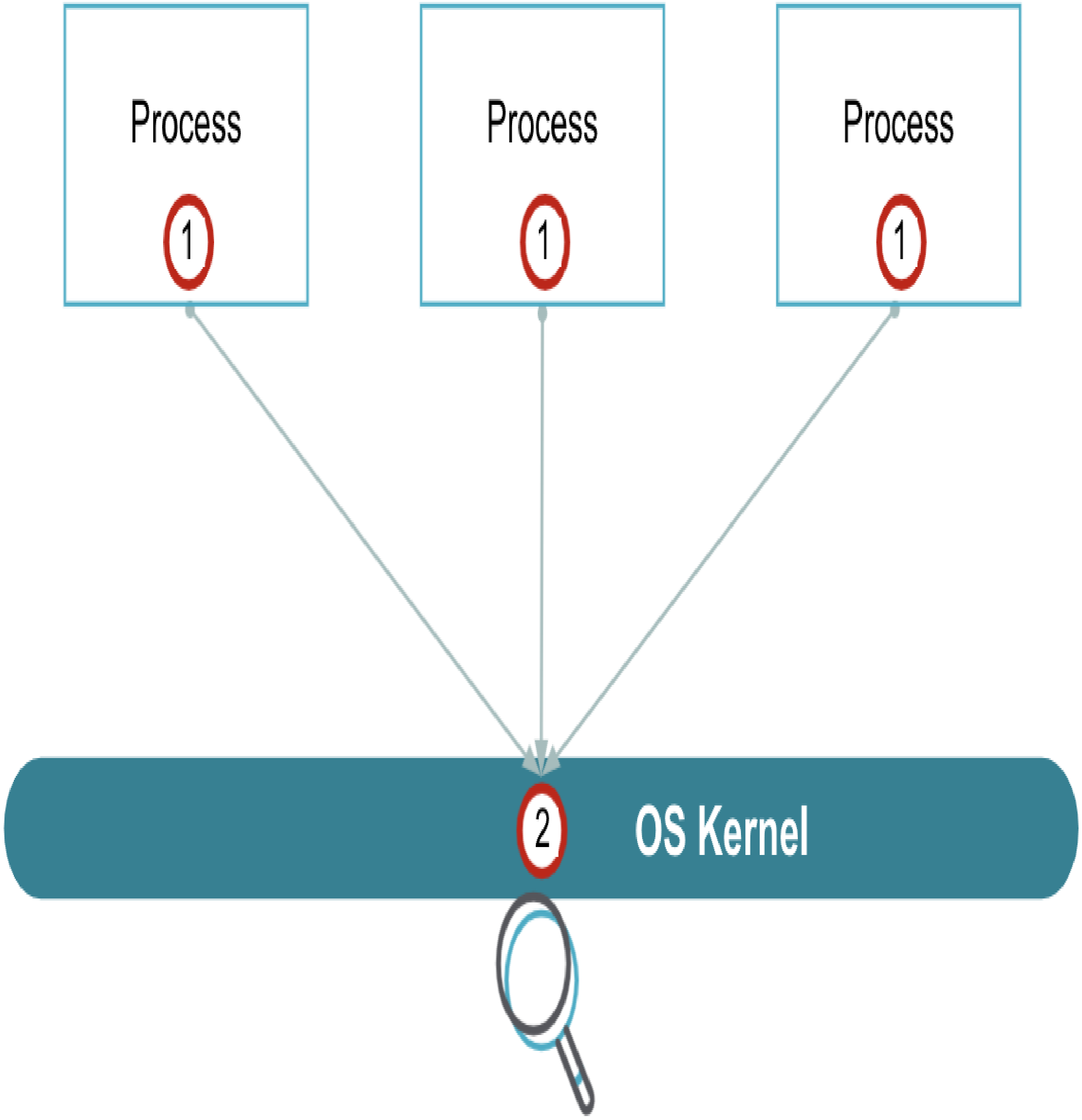


Figure 4-2. System calls capture options

If you insert kernel instrumentation in the right place, it is possible to have one single instrumentation point (labeled 2 in Figure 4-2), no matter how many processes are running. This means maximum efficiency, but also the certainty that you will never miss anything, because no process escapes kernel-level capture.

So What About Stability and Security?

We mentioned that kernel-level instrumentation is more delicate, because a bug can cause serious problems. You might wonder: “Am I taking additional risk by choosing a tool like Falco, which is based on kernel instrumentation, instead of a product based on user-level instrumentation?”

Not really. First of all, kernel-level instrumentation benefits from well-documented, stable hooking interfaces, while approaches like glibc-based capture are less clean and intrinsically riskier. They cannot crash the machine, but they can absolutely crash the instrumented process, with results that are typically bad. In addition to that, technologies like eBPF greatly reduce the risk involved in running code in the kernel, making kernel-level instrumentation viable even for risk-averse users.

Kernel-Level Instrumentation Approaches

We hope we’ve convinced you that, whenever it’s available, kernel instrumentation is the way to go for runtime security. The question now becomes: what is the best mechanism to implement it?

Among the different available approaches, two are relevant for a tool like Falco: kernel modules or eBPF probes. Let’s learn more about each of these approaches.

Kernel modules

Loadable kernel modules are pieces of code that can be loaded into the kernel at runtime. Historically, modules have been heavily used in Linux (and many other operating systems) to make the kernel extensible, efficient, and smaller.

Kernel modules extend the kernel’s functionality without the need to reboot the system. They are typically used to implement device drivers, network protocols, and file systems. Kernel modules are written in C and are compiled for the specific kernel inside which they will run. In other words, it’s not possible to compile a module on one machine and then use it on another one (unless they have exactly the same kernel). Kernel modules can also be unloaded when the user doesn't need them anymore, to save memory.

Linux has supported kernel modules for a very long time, so they work even with very old versions of Linux. They also have extensive access to the kernel, which means there are very few restrictions on what they can do. That makes them a great choice to collect the detailed information required by a runtime security tool like Falco. Since they are written in C, kernel modules are also very efficient, and therefore a great option when performance is important.

If you want to see the list of modules that are loaded in your Linux box, use this command:

```
$ sudo lsmod
```

eBPF

As mentioned in Chapter 1, eBPF is the "next generation" of the Berkeley Packet Filter (BPF). BPF was designed in 1992 for network packet filtering with BSD operating systems, and it is still used today by tools like Wireshark. BPF's innovation was the ability to execute arbitrary code in the kernel of the operating system. Since running code in the kernel means more or less unlimited privileges on the machine, executing arbitrary code is potentially risky and must be done with care.

Figure 4-3 shows how BPF safely runs arbitrary packet filters in the kernel.

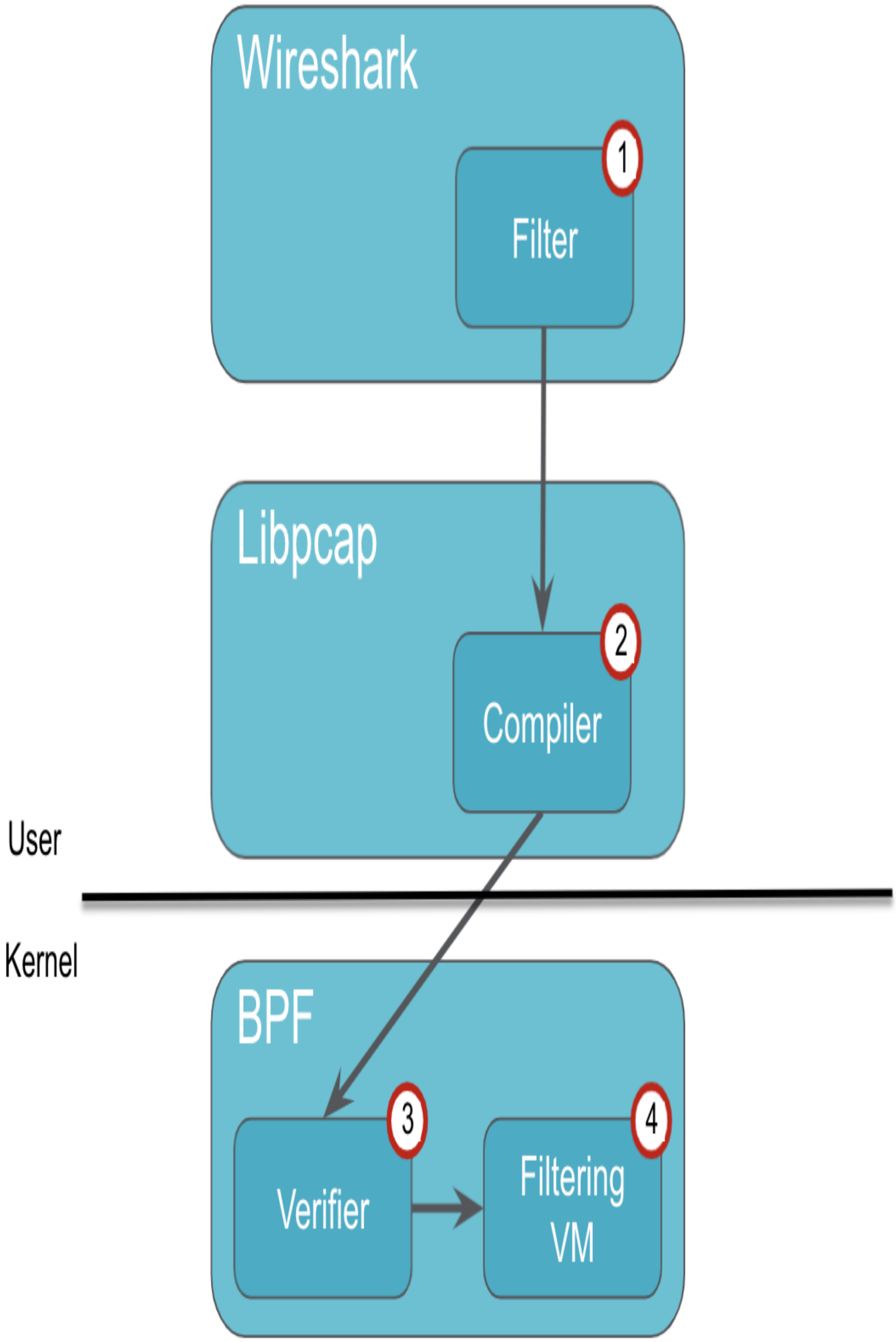

Wireshark
Filter
1
Libpcap
Compiler
2
User
Kernel
BPF
Verifier
3
Filtering VM
4

Figure 4-3. BPF filter deployment steps

Let’s take a look at the steps depicted in Figure 4-3:

1. The user inputs a filter in a program like Wireshark, e.g. `port 80`
2. The filter is fed to a compiler, which converts it into bytecode for a virtual machine. This is conceptually similar to compiling a Java program, but both the program and the virtual machine (VM) instruction set are much simpler when using BPF. Here, for example, is what our `port 80` filter becomes after being compiled:

```
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2    jf 10
(002) ldb      [20]
(003) jeq      #0x84            jt 6    jf 4
(004) jeq      #0x6             jt 6    jf 5
(005) jeq      #0x11            jt 6    jf 23
(006) ldh      [54]
(007) jeq      #0x50            jt 22   jf 8
(008) ldh      [56]
(009) jeq      #0x50            jt 22   jf 23
(010) jeq      #0x800           jt 11   jf 23
(011) ldb      [23]
(012) jeq      #0x84            jt 15   jf 13
(013) jeq      #0x6             jt 15   jf 14
(014) jeq      #0x11            jt 15   jf 23
(015) ldh      [20]
(016) jset     #0x1fff          jt 23   jf 17
(017) ldxb     4*([14]&0xf)
(018) ldh      [x + 14]
(019) jeq      #0x50            jt 22   jf 20
(020) ldh      [x + 16]
(021) jeq      #0x50            jt 22   jf 23
```

```
(022) ret      #262144
(023) ret      #0
```

3. To prevent a compiled filter from doing damage, it is analyzed by a verifier before being injected into the kernel. The verifier examines the bytecode and determines if the filter has dangerous attributes (for example, infinite loops that would cause the filter to never return, consuming a lot of kernel CPU).

4. If the filter code is not safe, the verifier rejects it, returns an error to the user, and stops the loading process . If the verifier is happy, the bytecode is delivered to the virtual machine, which runs it against every incoming packet.

eBPF is the young (and much more capable) child of BPF. eBPF was added to Linux in 2014 and was first included with kernel version 3.18. eBPF takes BPF's concepts to new levels, delivering more efficiency and taking advantage of newer hardware. Most importantly, with hooks all over the kernel, eBPF enables use cases that go beyond simple packet filtering, such as tracing, performance analysis, debugging, and security. It's essentially a general-purpose code-execution virtual machine that guarantees the programs it runs won't cause damage.

Here are some of the improvements that eBPF introduces over classic BPF:

- A more advanced instruction set, which means eBPF can run much more sophisticated programs.

- A just-in-time (JIT) compiler. While classic BPF was interpreted, eBPF programs, after being validated, are converted into native CPU instructions. This means they run much faster, close to native CPU speeds.

- A mature toolchain of C compiler: with eBPF you can write real C programs instead of just simple packet filters.

- A mature set of libraries that let you control eBPF from languages like Go.
- The ability to run subprograms and helper functions.
- Safe access to several kernel objects; eBPF programs can safely "peek" into kernel structures to collect information and context, which are gold for tools like Falco.
- The concept of *maps,* memory areas that can be used to exchange data with the user level efficiently and easily.
- A much more sophisticated verifier, which lets eBPF programs do "more" while preserving their safety.
- The ability to run in many more places in the kernel than the network stack, using facilities like Tracepoints, Kprobes, Uprobes, Linux Security Modules hooks, and USDT.

eBPF is evolving quickly and is rapidly becoming the standard way to extend the Linux kernel. While scripts are flexible and safe, they run extremely fast, making them perfect for capturing runtime activity.

The Falco Drivers

Falco offers two different capture driver implementations that implement both the approaches we just described: a kernel module and an eBPF probe. The two implementations have the same exact functionality and are interchangeable when using Falco. Therefore, we can describe how they work without focusing on a specific one.

The high-level capture flow is shown in Figure 4-4.

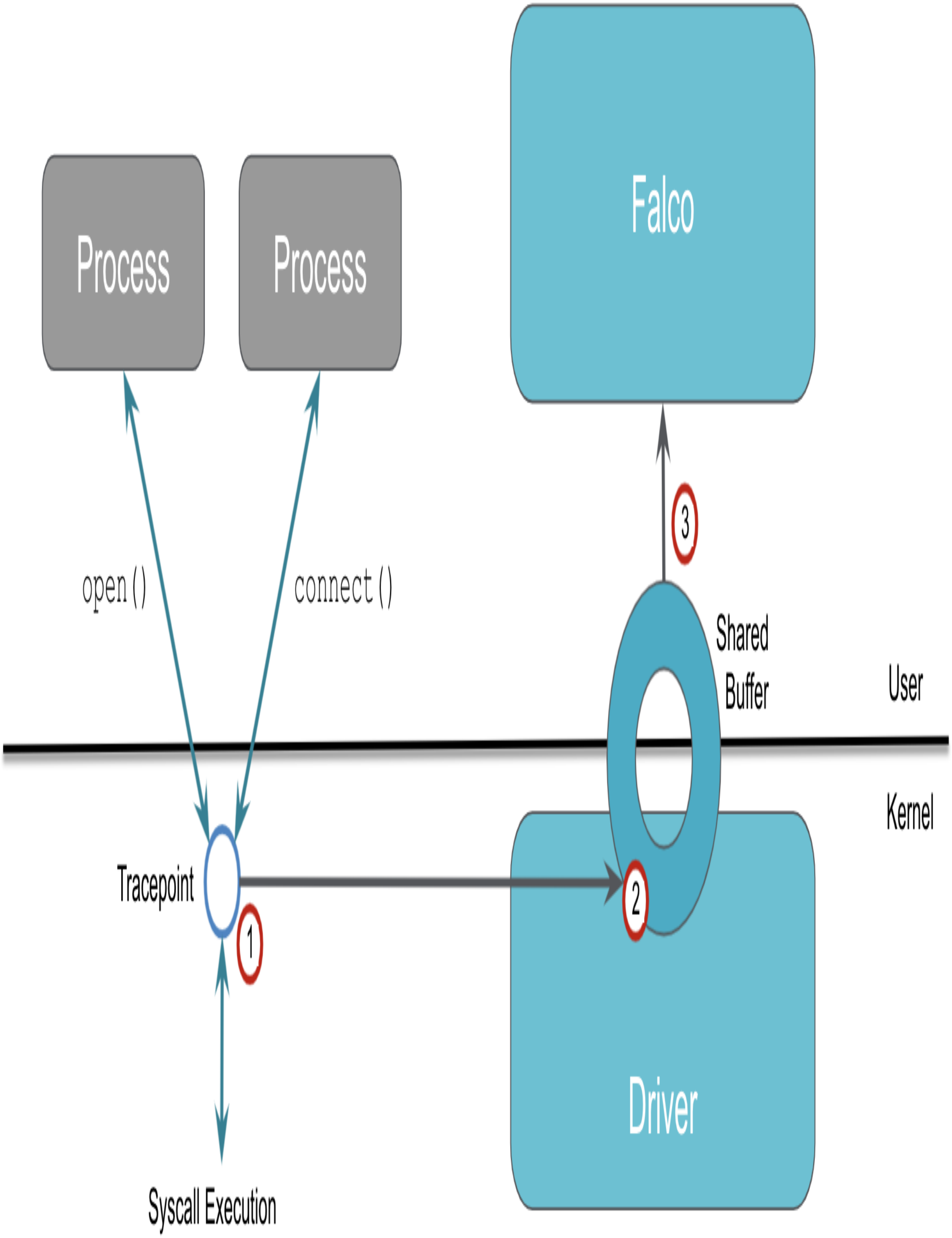


Figure 4-4. System calls capture

The approach used by the Falco drivers to capture a system call involves three main steps, labeled in Figure 4-4:

1. A kernel facility called a *tracepoint* intercepts the execution of the system call. The tracepoint makes it possible to insert a “hook” at a specific place in the operating system kernel, so that a callback function will be called every time that kernel execution reaches that point. For more information, see Mathieu Desnoyer’s article “Using the Linux Kernel Tracepoints.” The Falco kernel module and eBPF probe install two tracepoints for system calls: one where system calls enter the kernel, and another one where they exit the kernel and give control back to the caller process.

2. While in the tracepoint callback, the driver “packs” the system call arguments into a shared memory buffer. During this phase, the system call is also timestamped and additional context is collected from the operating system (for example, the thread ID, or the connection details for some socket syscalls). This phase needs to be super-efficient, because the system call cannot be executed until the driver’s tracepoint callback returns.

3. The shared buffer now contains the system call data, and Falco can access it directly through libscap (see Chapter 3). No data is copied during this phase, which minimizes CPU utilization while optimizing cache coherency.

There are some things to keep in mind with regards to system call capture in Falco.

The first one is that the way system calls are packed in the buffer is flexible and doesn’t necessarily reflect the arguments of the original system calls. In some cases, the driver skips unneeded arguments to maximize performance. In other cases, the driver adds fields that contain state, useful context, or additional information. For example, a `clone` event in Falco contains many fields that add information about the newly created process, like the environment variables.

The second thing to keep in mind is that, even if system calls are by far the most important source of data that the drivers capture, they are not the only ones. Using tracepoints, the drivers hook into other places in the kernel, like the scheduler, to capture context switches and signal deliveries. Take a look at this command:

```
sysdig evt.type=switch
```

This line of code displays events captured through the context switch tracepoint.

Which Driver Should You Use?

In case you are not sure which driver you should use, here are some simple guidelines.

Use the kernel module when you have an I/O intensive workload and you care about keeping the instrumentation overhead as low as possible. The kernel module overhead is lower than the one of the eBPF probe and, on machines that generate a high number of system calls, the kernel module will slow running processes less. It’s not easy to estimate much faster the kernel module will perform, since this depends on how many system calls a process is doing, but expect it to be noticeable with disk or network intensive workloads, which generate many system calls every second.

You should also use the kernel module when you need to support a kernel older than Linux version 4.12.

Use the eBPF probe in all other situations. That’s it!

Capturing System Calls Within Containers

The beauty of tracepoint-based kernel-level capture is that it sees everything that runs in a machine, inside or outside a container. Nothing escapes it. It is also easy to deploy, with no need to run anything inside the monitored containers. It also doesn’t require sidecars.

Figure 4-5 shows how you deploy Falco in a containerized environment, with a simplified diagram of a machine running three containers (labeled 1, 2, and 3) based on different container runtimes.

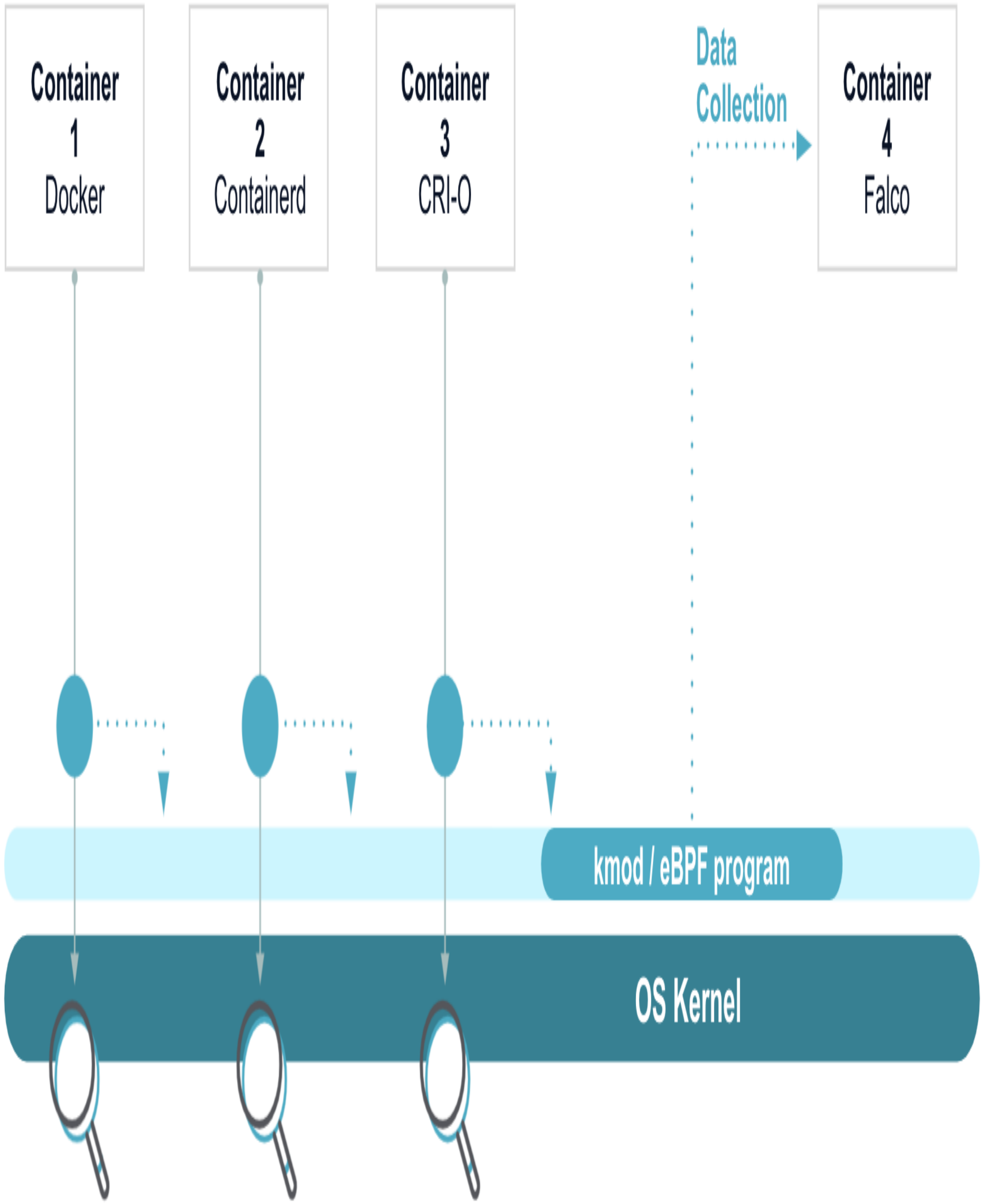


Figure 4-5. Deploying Falco in a containerized environment

In such a scenario, Falco is typically installed as a container. Orchestrators like Kubernetes make it easy to deploy Falco on every host, with facilities like daemonsets and helm charts.

When the Falco container starts, it installs the driver in the operating system. Once installed, the driver can see the system calls of any process in any container, with no further user action required, because all of these system calls go through the same tracepoint. Advanced logic in the drive can attribute each captured system call to its container, so that Falco always knows which container has generated a system call. Falco also fetches metadata from the container runtime, making it easy to create rules that rely on container labels, image names, and other metadata. (Falco includes a further level of enrichment based on Kubernetes metadata, but we will focus on that in the next chapter.)

Running the Falco Drivers

Let’s learn how to deploy and use the two Falco drivers on a local machine. (If you want to install Falco in production environments, go to Chapters 8 and 9.)

Kernel Module

Falco, by default, runs using the kernel module, so no additional steps are required if you want the kernel module to be the driver. Just run Falco and it will pick up the kernel module. If you want to unload the kernel module and load a different version, for example because you have built your own customized version, use the following commands:

```
$ sudo rmmod falco
$ sudo insmod /kernel/module/path/falco.ko
```

EBPF PROBE

If you are using the eBPF probe to ensure that performance is not degraded, make sure that your kernel has `CONFIG_BPF_JIT` enabled and that `net.core.bpf_jit_enable` is set to `1 (enable the BPF JIT Compiler).`

To enable eBPF support in Falco, you need to set the `FALCO_BPF_PROBE` environment variable. If you set it to an empty value (`FALCO_BPF_PROBE=""`), Falco will load the eBPF probe from `~/.falco/falco-bpf.o`.

Otherwise, you can explicitly point to the path where the eBPF probe resides (`FALCO_BPF_PROBE="/ebpf/probe/path/falco-bpf.o"`). After setting the environment variable, just run Falco normally and it will use the eBPF probe.

Using Falco in Environments with No Kernel Access: pdig

Kernel instrumentation, whenever possible, is always the way to go. But what if you want to run Falco in environments where access to the kernel is not allowed? This is common in managed container environments, like Fargate in AWS. In such environments, installing a kernel module is not an option because the cloud provider blocks it.

For these situations, Falco developers have implemented a user-level instrumentation driver called pdig. It is built on top of ptrace, so it uses the same approach as strace. Like strace, pdig can operate in two ways: It can run a program that you specify on the command line, or it can attach to a running process. Either way, pdig instruments the process and its children in a way that produces a Falco-compatible stream of events.

Note that pdig, like strace, requires you to enable `CAP_SYS_PTRACE` for the container runtime. Make sure you launch your container with this capability, or pdig will fail.

The eBPF probe and kernel module work at the global host level, whereas pdig works at the process level. This can make container instrumentation more challenging. Fortunately, pdig can track the children of an instrumented process. This means that running the entrypoint of a container with pdig will allow you to capture every system call generated by any process for that container.

The biggest limitation of pdig is performance. ptrace is versatile, but it introduces substantial overhead on the instrumented processes. pdig employs several tricks to reduce this overhead, but it's still substantially slower than the kernel-level Falco drivers.

Running Falco with pdig

Run pdig with the path (and arguments, if any) of the process you want to trace, much as you would with `strace`. Here is an example:

```
$ pdig [-a] curl https://example.com/
```

The `-a` option enables the full filter, which provides a richer set of instrumented system calls. You probably don't want to use this option with Falco for performance reasons.

You can also attach to a running process with the `-p` option:

```
$ pdig [-a] -p 1234
```

To observe any effect, you will need to have Falco running in a separate process. Use the `-u` command line flag:

```
$ falco -u
```

This will enable user space instrumentation.

Falco Plugins

Let’s finish this chapter by learning how Falco collects and processes logs and cloud APIs: through its plugins framework.

Plugins are a modular, flexible way to extend Falco ingestion. Anyone can use them to add a new source of data, local or remote, to Falco. Figure 4-6 gives an overview of where plugins sit in the Falco capture stack: they are inputs for libscap and act as alternatives to the drivers that are used when capturing system calls.

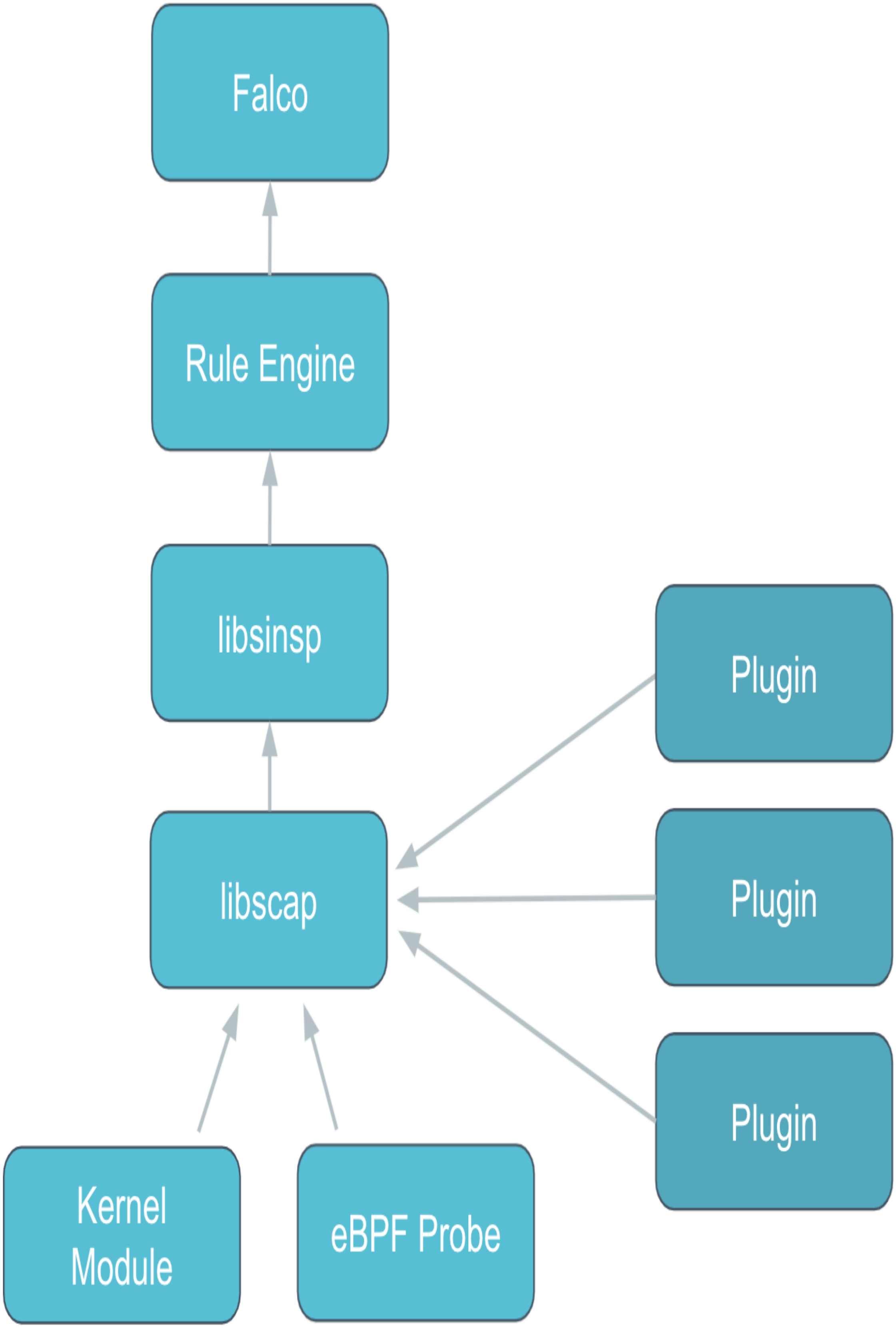

Falco
Rule Engine
libsinsp
libscap
Plugin
Plugin
Plugin
Kernel Module
eBPF Probe

Figure 4-6. Falco plugins

Plugins are implemented as shared libraries that conform to a documented API. They allow you to add new event sources that you can then evaluate using filtering expressions and Falco rules. They also let you define new fields that can extract information from events.

Plugin Architecture Concepts

Plugins are dynamic shared libraries (.so files in Unix, .dll files in Windows) that export C calling-convention functions. Falco dynamically loads these libraries and calls the exported functions. Plugins are versioned using semantic versioning to minimize regressions and compatibility issues. They can be written in any language, as long as they export the required functions. Go is the preferred language for writing plugins, followed by C/C++. There are two kinds of plugins:

Source Plugins

> A source plugin provides a new scap event source. It can “open” and “close” a stream of events and can return an event to libscap via a `next()` method.
>
> Optionally, source plugins are also capable of extracting fields from the events they generate. *Fields*, you’ll recall, are the basic components of Falco rules, so exposing new fields is equivalent to expanding the applicability of Falco rules to new domains. We’ll learn more about fields in Chapter 6.

Extractor Plugins

> An extractor plugin focuses only on field extraction from events generated by other plugins or by the core libraries. It does not provide an event source, but can extract fields from other event sources. An example is json field extraction, where a plugin might be able to extract fields from arbitrary json payloads.

To make it easier to write plugins, there are Go and C++ software developer's kits (SDKs) that handle the details of memory management and type conversion. They provide a streamlined way to implement plugins without having to deal with all the details of lower-level functions that make up the plugin API.

The libraries will do everything possible to validate data that comes from the plugins, to protect Falco and other consumers from corrupted data. However, for performance reasons, plugins are "trusted": they run in the same thread and address space as Falco, so they *could* crash the program. Falco assumes that you, as a user, are in control and will make sure only trusted plugins are loaded or packaged.

How Falco Uses Plugins

Falco loads plugins based on the configuration in falco.yaml. If a source plugin is configured, the only events processed are from that plugin and syscalls capture is disabled. Also, a running Falco instance can use a single plugin. If, on a single machine, you want Falco to collect data from multiple plugins or from plugins and drivers, you will need to run multiple Falco instances and use a different source for each of them.

Falco configures plugins via the `plugins` property in falco.yaml. Here's an example:

```
plugins:
  - name: aws_cloudtrail
    library_path: aws_cloudtrail/plugin.so
    init_config: "..."
    open_params: "..."
  - name: http_json
    library_path: http_json/plugin.so
    init_config_file: http_json/config.txt
    open_params_file: http_json/params.txt

# Optional
load_plugins: [aws_cloudtrail]
```

The `plugins` property in falco.yaml defines the set of plugins that Falco can load, as well as the `load_plugins` property, which controls which plugins load when Falco starts.

The mechanics of loading a plugin are implemented in libscap and leverage the dynamic library functionality of the operating system (dlopen/dlsym in Unix, LoadLibrary/GetProcAddress in Windows). The plugin loading code also ensures that:

- The plugin is valid: it exports the set of expected symbols
- The plugin's API version number is compatible with the plugin framework
- Only one source plugin is loaded at a time for a given event source
- If a mix of source and extractor plugins is loaded for a given event source, the exported fields have unique names that don't overlap across plugins

An up-to-date list of available Falco plugins can be found in the plugins repository under the Falco github organization. As of this writing, available plugins include Cloudtrail and GitHub.

If you are interested in writing your own plugins, you will find everything you need to know in chapter 14. If you are impatient and just want to get to the code, you can find the source code of several plugins at Falco's GitHub organization.

Conclusion

Congratulations for making it to the end of a rich chapter packed with a lot of information! What you learned is at the core of understanding and operating Falco. It also constitutes a solid architectural foundation that will be useful every time you need to run or deploy a security tool on Linux.

Next, we are going to learn about how context is added to the captured data to make Falco even more powerful.

[1] Run `man 2 ptrace` for more information.

Chapter 5. Data Enrichment

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Falco's architecture allows you to capture events from different data sources, as you've learned. This process delivers raw data—which can be very rich, but isn't very useful for runtime security unless paired to the right context. That's why Falco extracts and then enriches the raw data with contextual information so that the rule author can comfortably use it. Typically, we refer to this information as the event *metadata*. Getting metadata can be a complex task; getting it efficiently is even more complex.

You have already seen that the system-state collection capabilities in libscap (discussed in Chapter 4) and the state engine implemented by libsinsp (Chapter 3) are central to this activity. But there is much more to discover.

In this chapter, we delve into the design aspects of the Falco stack to help you better understand how data enrichment works. In particular, you may need information relating to different contexts, depending on your use case, such as a container's ID or the name of a pod where a suspicious event occurred. For this reason, we will show you libsinsp's efficient layered approach to obtaining system, container, and Kubernetes metadata for system call (syscall) events. Finally, we'll show you how plugins, Falco's other main data source, can implement their own data enrichment mechanisms, opening the way to infinite possibilities.

Understanding Data Enrichment in Falco

Understanding how data enrichment works will help you to fully understand Falco's mechanics. Moreover, although data enrichment usually works out-of-the-box, each context Falco supports has its own implementation and may need a specific configuration. Knowing the implementation details will help you troubleshoot and fine-tune Falco.

Data enrichment, in Falco, refers to the process of providing the rule engine with event metadata. Falco's data enrichment mechanism obtains metadata by decoding the raw data or collecting it from complementary sources, to be used later in filters and output.

Thanks to this process, you can use the resulting metadata as fields in both rule conditions and output formatting. Furthermore, Falco organizes the collected metadata in a set of field classes, so you can easily recognize which context they belong to. (You can find the complete list of supported fields in Chapter 6 or, if you have a Falco installation at your fingertips, by typing: `falco --list`.)

One of the most significant examples of data enrichment is when using system calls as a data source, which you learned about in Chapter 4. Since syscalls are essential to every application, they occur in just about every context. Information directly provided by a syscall would not be useful without context. Thus it becomes critical to collect and connect the surrounding information. Table 5-1 shows the different categories of metadata that Falco collects for syscalls.

Table 5-1. Contextual metadata for

system calls

Contexts	Metadata	Field classes
System	Processes and threads File descriptors Users and groups Network interfaces	`proc`, `thread`, `fd`, `fdlist`, `user`, `group`
Container	ID and name Type Image name Privileged Mounts points Health checks	`container`
Kubernetes	Namespace Pod Replication Controller Service Replica Set Deployment	`k8s`

The enrichment process happens in userspace and involves several components of Falco’s stack. Most importantly, the metadata must be immediately available every time the rule engine requests it. Collecting it from other complementary sources on the fly would not be feasible: attempting to do so would risk blocking the rule engine and the entire flow of incoming events.

For that reason, the data enrichment design includes two distinct phases. The first initializes a local state by collecting in bulk the data that is present when Falco starts; the second continuously updates the local state while Falco runs. Having a local state allows Falco to extract metadata immediately. This design is shared among all implementation layers, as you will discover in the following sections.

KUBERNETES SUPPORT AND KUBERNETES AUDIT LOG ARE NOT THE SAME

In the documentation, you will often find mention of Kubernetes support and Kubernetes Audit Log. Sometimes, one might be confused and think they are the same thing. These are actually two distinct features.

The first, generally referred to as *Kubernetes support*, only concerns the enrichment of an event originating from a syscall with Kubernetes metadata. In rules that metadata is available through the `k8s` field class. That's what we talk about in this chapter.

On the other hand, Kubernetes Audit Log is an independent data source (not originating from a syscall). You can quickly identify rules that use this data source because they need to have the `source: k8s_audit`. The metadata is accessible through the `ka` field class. Furthermore, to make the Kubernetes Audit Log work, it must be enabled both on Kubernetes and Falco. In this latter case, Kubernetes directly feeds Falco with events, sending them via a webhook. Also, the Kubernetes Audit Log system already provides all the necessary context data along with the originating event, and therefore no specific enrichment mechanism is needed.

You can enable the two features separately since they are not dependent on each other.

System Metadata

As you learned in Chapter 3, libscap and libsinsp work together to provide all the necessary infrastructure to create and update contextual information in a hierarchical structure composed of several state tables (see Figure 3-4 if you need a refresher). Those tables include information about:

- Processes and threads

- File descriptors
- Users and groups
- Network interfaces

At a high level, the mechanism for collecting system information is relatively simple. At start time, one of libscap's tasks is to scan the process information pseudo-filesystem, or procfs, which provides a user-space interface to the Linux kernel data structures and contains most of the information to initialize the state tables. It also collects system information (not available in */proc*) using functions provided by the standard C library, which in turn obtains the data from the underlying operating system (for example, `getpwent` and `getgrent` for users and groups lists, respectively, and `getifaddrs` for the network interfaces list). At this point, the initialization phase is complete.

> **TIP**
>
> Libscap and libsinsp rely on the host's procfs to access the host's system information. That happens by default when Falco runs on the host since it can directly access the host's */proc*. However, when Falco runs in a container, the */proc* inside the container refers to a different namespace. In such a situation, you can configure libscap via the `HOST_ROOT` environment variable to read from an alternate path. If you set `HOST_ROOT` , libscap will use its value as a base path when looking for system paths. For example, when running Falco in a container, the usual approach is to mount the host's */proc* to */host/proc* inside the container and set `HOST_ROOT` to */host*. With this setup, libscap will read from */host/proc*, and thus it will use the information provided by the host's procfs.

Afterward, libsinsp comes into play with its state engine. It updates the tables by inspecting the constantly captured stream of syscalls provided by the driver, which runs in kernel space. After the initialization phase, Falco will not need to make any syscalls nor tap into the system to obtain updates from the Linux Kernel. This approach has the double benefit of not creating noise in the system and having a low impact on performance. Furthermore, this technique enables libsinp to discover system changes with low latency, allowing Falco to function as a streaming engine, one of its most important design goals.

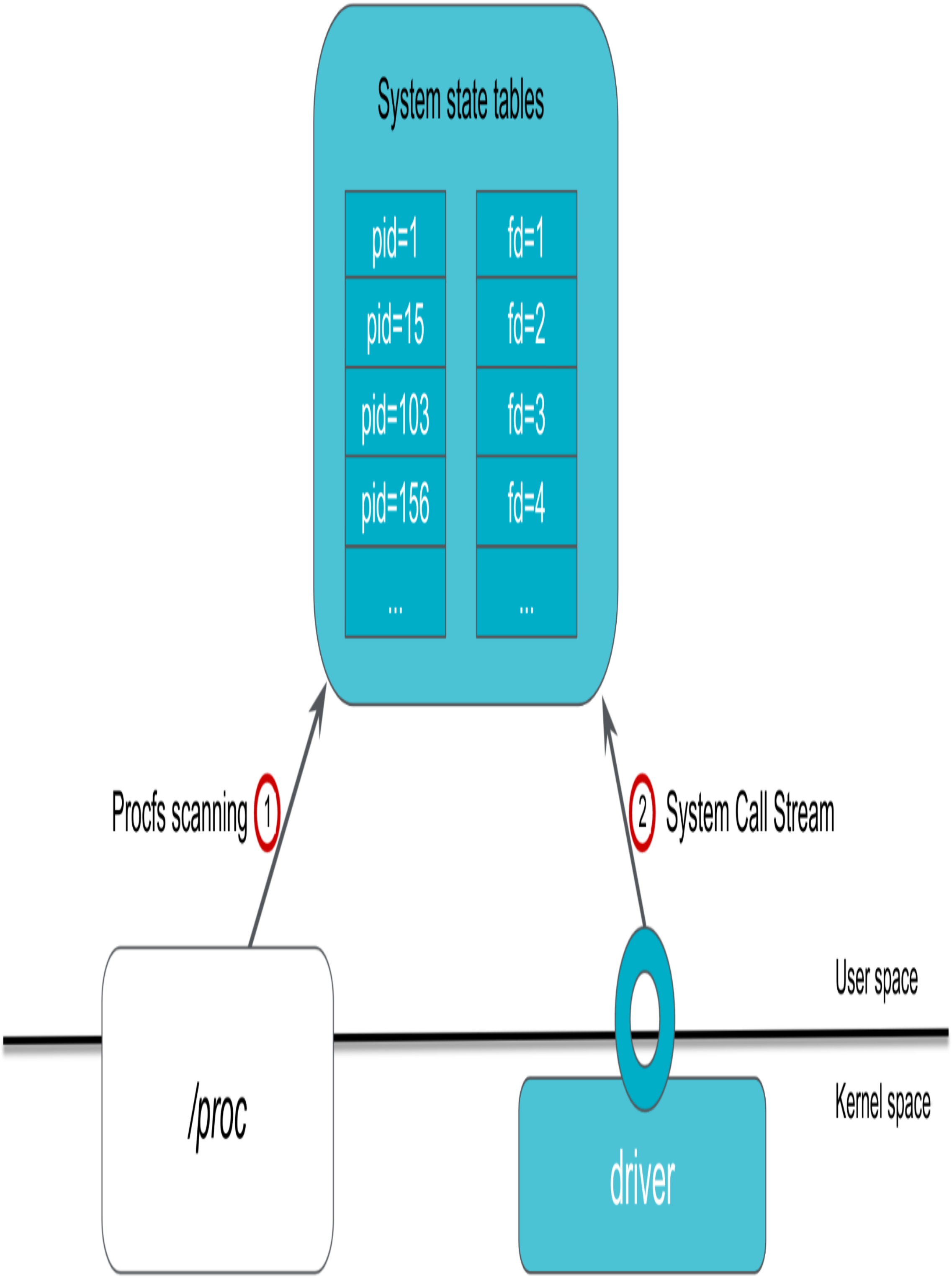

System state tables
pid=1
pid=15
pid=103
pid=156
...
fd=1
fd=2
fd=3
fd=4
...
Procfs scanning
1
2
System Call Stream
User space
Kernel space
/proc
driver

Figure 5-1. System state collection before (1) and after (2) the initialization phase

The last important note is that libsinsp updates the state tables before dispatching the event to the rule engine. This ensures that when the conditions or output require metadata, it will always be available and consistent. You can then find the system metadata grouped in the set of field classes you saw in Table 5-1: `proc`, `thread`, `fd`, `fdlist`, `user`, `group`.

This set of information represents the basic metadata that enables a rule author to make a syscall event usable. Think about it: how would you use a numeric file descriptor in a rule? A file name is much better!

The state tables containing system information are also a fundamental requirement of another data enrichment layer. Let’s look at that next.

Container Metadata

Another fundamental piece of context information resides in the container runtime layer. A *container runtime* is a software component that can run containers on a host operating system. It is commonly responsible for managing container images and the lifecycles of containers running on your system. It is also responsible for managing a set of information related to a running container and providing that information to other applications.

Since, by definition, Falco is a cloud-native runtime security tool, dealing with container information is one of its most important aspects. To achieve this goal, libsinsp works with the most used container runtime environments, including Docker, Podman, and any CRI-compatible runtime, like containerd and CRI-O.[1]

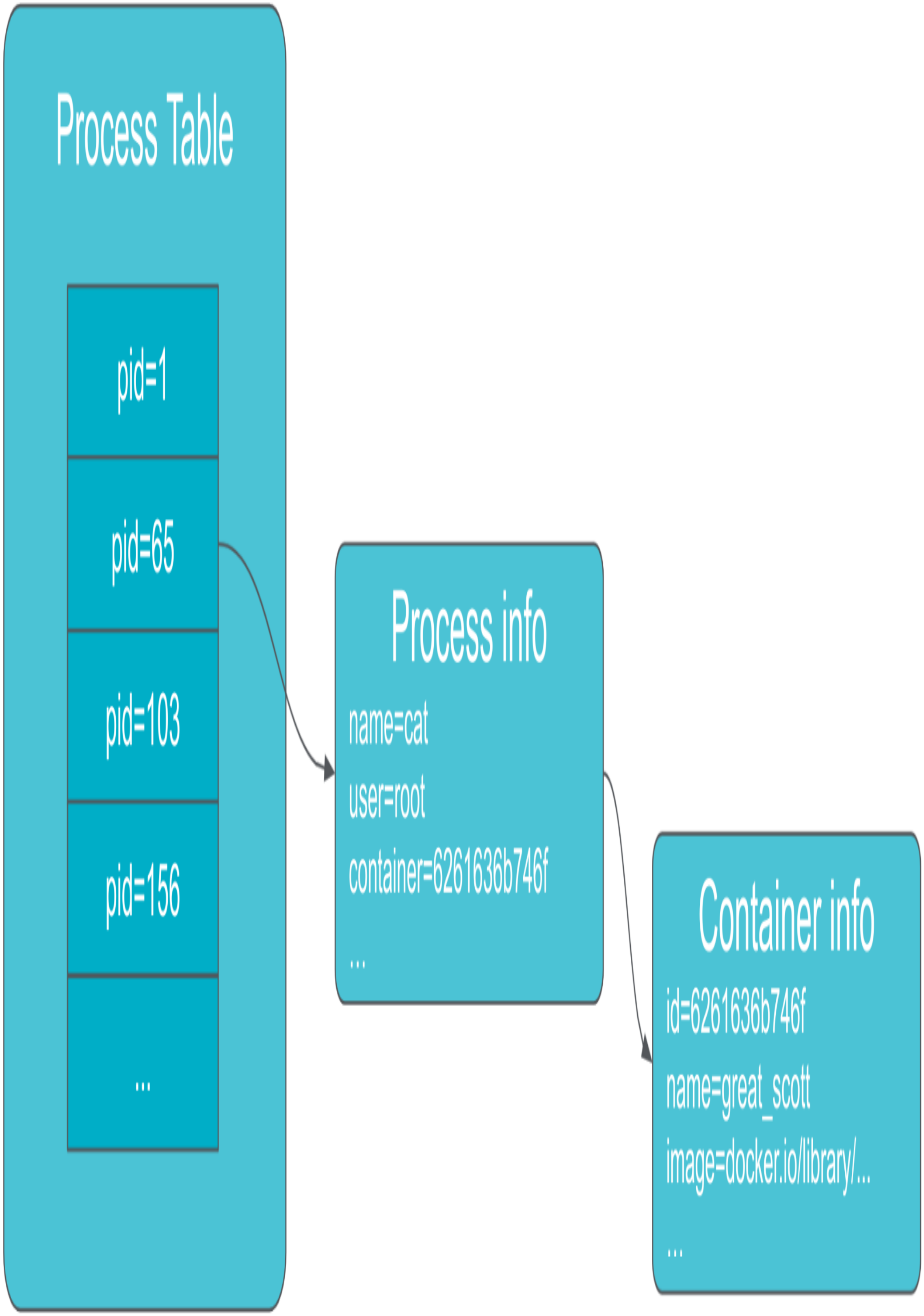
When libsinsp finds a running container runtime on the host, the container metadata enrichment works out-of-the-box in almost all cases. For example, libsinsp tries to use Docker’s Unix socket at */var/run/docker.sock*; if this exists, libsinsp automatically connects to it and starts grabbing container metadata. However, libsinsp requires the user to specify the path to the

Unix socket for CRI-compatible runtimes. So, if you want to use a CRI runtime, you will need to pass the socket path to Falco using the `--cri` command line flag (for containerd, for example, you would pass */run/containerd/containerd.sock*).

> **TIP**
>
> If the `HOST_ROOT` environment variable is set, libsinsp will use its value as the base path when looking for those Unix sockets. For example, when running Falco in a container, it’s common to set `HOST_ROOT=/host` and mount */var/run/docker.sock* to */host/var/run/docker.sock* inside the container.

Regardless of which container runtime you are using, at initialization, libsinsp requests a list of all running containers, which it uses to initialize an internal cache. At the same time, libsinsp updates the state table of running processes and threads, associating each of them with its respective container ID, if any. Libsinsp handles subsequent updates by using the syscalls stream coming from the driver (similar to what it does for system information). Since container information is always associated with a process, libsinsp tracks all new processes and threads. When it detects one, it looks up the corresponding container ID from the internal cache. If the container ID is not in the cache, libsinsp queries the container runtime to gather the missing data.

> **TIP**
>
> The process of querying the container runtime happens asynchronously to avoid blocking the stream of events. In some environments, this operation is not fast enough to be completed asynchronously, so attempting it leads to empty container metadata fields. For CRI-compatible runtime, Falco provides an option to disable asynchronous metadata fetching: `--disable-cri-async`.
>
> Although you won’t generally need to disable this, it is helpful to let the input event wait for the container metadata fetch to finish before moving forward, so that no metadata is lost. However, you might see a performance penalty depending on the number of containers and the frequency with which they are created, started, and stopped.



Process Table
pid=1
pid=65
pid=103
pid=156
...
Process info
name=cat
user=root
container=6261636b746f
...
Container info
id=6261636b746f
name=great_scott
image=docker.io/library/...
...

Figure 5-2. Container info in the libsinsp state hierarchy

In the end, each syscall-generated event that occurs in a container has a process or a thread ID that maps to a container ID and, consequently, to the container metadata (as shown in Figure 5-2). So, when the rule engine requires this metadata, libsinsp looks it up from the state tables and returns system information along with the container metadata. You will find the available container metadata grouped in the field class `container`, which can be used in both condition and output formatting.

Note that the field `container.id` can contain either the container ID *or* the special value `host`. This special value indicates that the event did not happen inside a container. The condition `container.id != host` is a common way to express a rule that only applies in the context of a container.

Libsinsp provides another data enrichment layer that introduces a new fundamental context in cloud environments: Kubernetes metadata.

Kubernetes Metadata

Kubernetes is an open source platform for managing workloads and services and is also the flagship project of the Cloud Native Computing Foundation. Nowadays, it is the most popular container orchestration system. Kubernetes has introduced many new concepts in cluster management that make it easier to manage and scale.

One of Kubernetes' essential features is encapsulating your applications in objects called *Pods* (which use containers inside them). Pods are ephemeral objects that you can quickly deploy and easily replicate. On the other hand, *Services* are an abstraction that allows you to group a set of pods together logically if they exhibit the same function. Finally, Kubernetes lets you arrange those and many other objects into *Namespaces*, which are other objects for partitioning a single cluster into multiple virtual clusters.

While these concepts greatly facilitate managing and automating clusters, they also introduce a set of contextual information about how and where your application is running. This is essential, since knowing that something has happened in your Kubernetes cluster is of little use if you don't know where it happened (for example, in which namespace or pod). Falco collects information such as the container image name, pod name, namespace, service labels, and annotations so it can offer as accurate a view as possible of your deployment and application. That is very important when doing runtime alerting and protection of your infrastructures, because you're typically much more interested in what service or deployment is showing a strange behavior than in getting a container ID or some other hard-to-link piece of information.

Since Falco was born as a cloud-native tool, it can readily obtain this metadata and attach it to the event. Similar to the system and container metadata mechanisms you saw in previous sections, this feature allows Falco to enrich syscall events by adding Kubernetes metadata. For full Kubernetes support, you must opt in by passing two command-line options to Falco. First:

`--k8s-api` (or just `-k`)

This enables Kubernetes support by connecting to the API server specified as an argument (e.g. *http://admin:password@127.0.0.1:8080*).

`--k8s-api-cert` (or just `-K`)

This provides certificate materials to authenticate the user and (optionally) verify the Kubernetes API server's identity.

Once Kubernetes support is configured, libsinsp will get all the necessary data from Kubernetes to create and maintain a local state of the cluster. However, unlike the other enrichment mechanisms that get metadata locally from the host, libsinsp has to connect to the Kubernetes API server (usually a remote endpoint) to get cluster information. Because of this difference, the implementation design needs to take performance and scalability concerns into account.

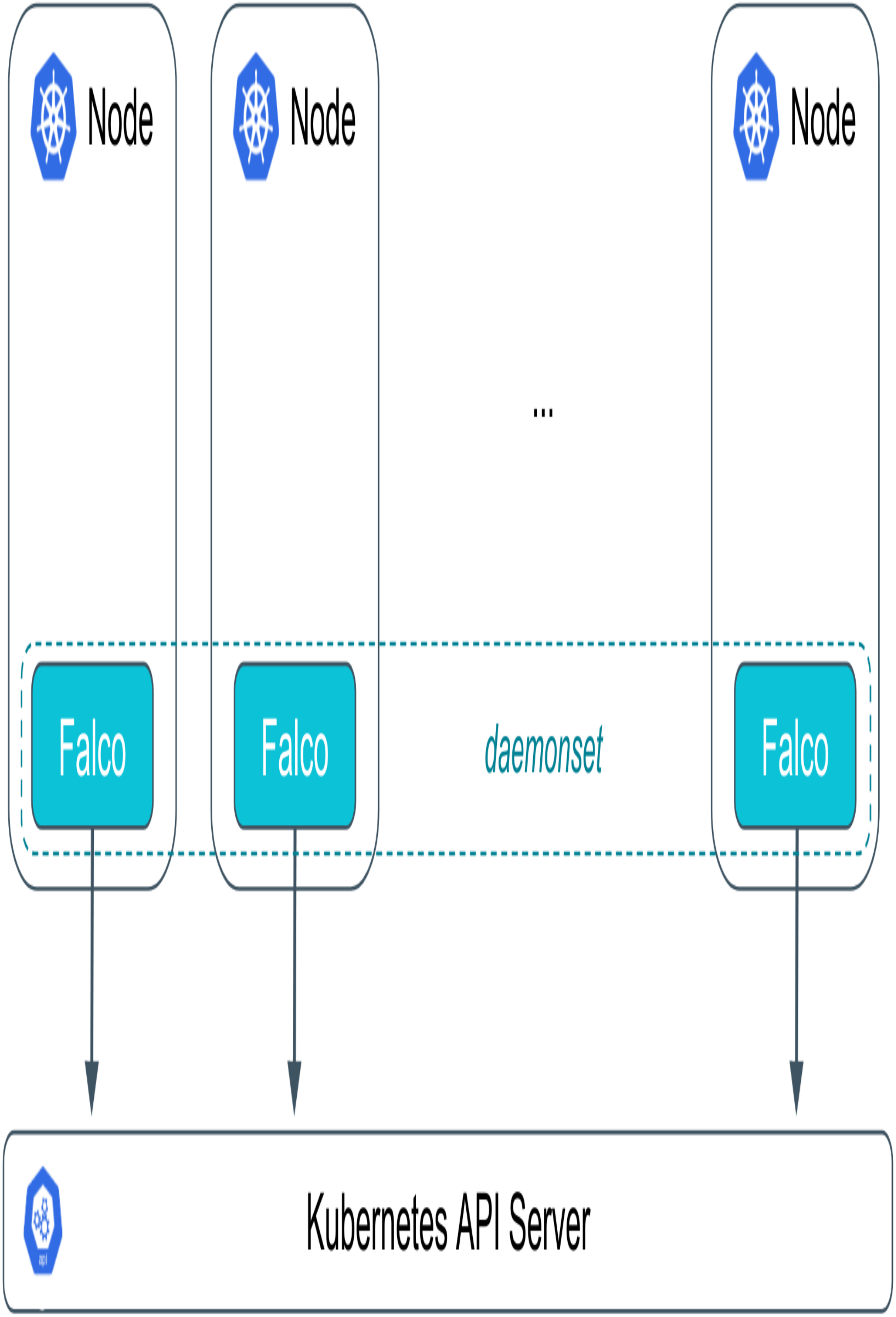

Node
Node
...
Node
Falco
Falco
daemonset
Falco
Kubernetes API Server

Figure 5-3. Falco deployment, using a DaemonSet to ensure that all Nodes run a copy of a Pod

A typical Falco deployment (pictured in Figure 5-3) runs one Falco sensor on every node in the cluster. At startup, each sensor connects to the server API to collect the cluster data and build the initial state locally. From then on, each sensor will use the Kubernetes watch API to periodically update the local state.

Since Falco sensors are distributed in the cluster (one per node) and grab data from the API server, libsinsp has mechanisms to avoid congestion. Moreover, collecting from Kubernetes some resource types may result in huge responses that severely impact both the API server and Falco. Libsinsp's straightforward strategy is to wait for a short time between downloading each chunk. Falco allows you to fine-tune that wait time, along with several other parameters, which you can find and change in */etc/falco/falco.yaml*.

Most importantly, it's possible to request *only* the relevant metadata for the targeted node from the server API. This is possible because Falco's architecture is distributed, so each sensor only needs data from the node on which the event occurred. This optimization is fundamental if you want to scale Falco on a cluster with thousands of nodes. To enable it, pass the `--k8s-node` flag to the Falco command-line arguments and value it with the current node name. You can usually obtain the current node name easily from the Kubernetes Downward API.[2]

If you do not pass the `--k8s-node` flag, libsinsp will still be able to get the data from Kubernetes, but each Falco sensor will have to request the whole cluster's data. This can introduce a performance penalty on large clusters, so we strongly discourage it. (You will learn more about running Falco on a production Kubernetes cluster in Part III.)

TIP

Although the method described in this section is the recommended way to obtain Kubernetes metadata, there’s another, leaner working mode worth mentioning. However, before teaching you how to use it, a bit of explanation is needed.

Libsinsp is smart enough to get a subset of the metadata even without connecting to the Kubernetes API server. That’s possible because the kubelet annotates some metadata directly on the container: specifically the ID, name, namespace, and labels of the pod (which are usually the most relevant context information). Since libsinsp retrieves those annotations using the container runtime API, it also tries to use them when possible and falls back to the Kubernetes API server when the missing data is needed. This strategy is always enabled so you will not need to configure it.

You can think of this strategy as an optimization, but also as a feature. Indeed, if that portion of the metadata is enough for your use case (for example, if you’re using a custom ruleset that doesn’t need the complete set of Kubernetes metadata), you don’t have to enable full support for Kubernetes. If you don’t, you will still get the ID, name, namespace, and labels of the pod.

When Kubernetes metadata is available, you will find it grouped in the `k8s` field class. Most of the Falco default rules include `k8s` fields in their condition also print the most crucial Kubernetes metadata in their output. Here is an example of notification output:

```
15:29:40.515013896: Notice System user ran an interactive command
(user=bin user_loginuid=-1 command=login
container_id=46c99eea62a8 image=docker.io/library/nginx)
k8s.ns=default k8s.pod=my-app-84d64cb8fb-zmxgz
container=46c99eea62a8
```

That output is the result of the complex mechanism you’ve just learned about, that allows you to obtain accurate and contextualized information to immediately identify what and where the event occurred.

So far, we have only discussed data enrichment for system calls. Although that’s likely to be the most relevant part about the metadata enrichment, you should know that Falco also offers custom enrichment mechanisms.

Data Enrichment with Plugins

Plugins can extend Falco by adding new data sources and defining new fields to describe how to use these new events. As you'll recall from Chapter 4, there are two kinds of plugins: source and extractor. The former uses its own data source as input, while the latter works on a data source provided by other plugins or core libraries.

While it might not seem obvious yet, the extractor plugin has all it takes to implement a data enrichment mechanism. First, it can receive data from any data source. Second, it can define new fields. Fundamentally, it allows the plugin author to implement logic to return the value of those fields, thus potentially providing additional metadata. This opens the door to the possibility of implementing custom data enrichment.

When the extractor plugin runs, libsinsp calls the plugin function for field extraction for each incoming event. The function receives the raw payload of the event and the list of fields the rule engine needs. The plugin API interface does not place any other constraints to make the extraction process work. Although data enrichment is possible in the flow just described, the plugin author will still have to consider all the use case's implications: for example, the plugin will need to manage the local state and subsequent updates. Extracting fields and enriching the event is thus entirely up to the plugin author. The APIs provide all the essential tools.

Chapter 14 shows you how to implement a plugin. Our advice, however, is to complete the next chapter about fields and filters first, so you can have a complete picture of how extracting data works.

[1] The Container Runtime Interface (CRI) is a plugin interface introduced by Kubernetes that enables the kubelet to use any container runtimes implementing the CRI.

[2] The Downward API allows containers to consume information about themselves or the cluster without using Kubernetes API server. Among other things, it allows exposing the current node name through an environment variable that can be then used in Falco command-line arguments.

Chapter 6. Fields and Filters

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

It’s finally time to take all the theory you learned in the previous chapters and start putting it into practice. In this chapter you will learn about Falco filters: what they are, how they work, and how to use them.

Filters are at the core of Falco. They are also a powerful investigation instrument that can be used in several other tools, for example Sysdig. As a consequence, we expect that you will come back and consult this chapter often, even after finishing the book–so we’ve structured it to be used as a reference. For example, it contains tables with all of the operators and data types the filtering language provides, designed for quick consultation, as well as a well-documented list of Falco’s most useful fields. It will be handy pretty much every time you write a Falco rule, so make sure to bookmark it!

What Is a Filter?

Let’s start from a semi-formal definition: A *filter* in Falco is a condition containing a sequence of comparisons that are connected by Boolean operators. Each of the comparisons evaluates a field, which is extracted from an input event, against a constant, using a relational operator. Comparisons in filters are evaluated left to right, but parentheses can be used to define precedence. A filter is applied to an input event and returns a Boolean result indicating if the event matches the filter.

Ouch. That sounded extremely dry and somewhat complicated. Let’s unpack it, with the aid of some examples, and you’ll see it’s not too bad. We can start with the first sentence:

> *A filter in Falco is a condition containing a sequence of comparisons that are connected by Boolean operators.*

This just means that a filter looks like this:

```
A = B and not C != D
```

In other words, if you can write an *if* condition in any programming language, the filter syntax will look very familiar.

> *Each of the comparisons evaluates a field, which is extracted from an input event, against a constant, using a relational operator.*

The Falco filtering syntax is based on the concept of *fields*, which we will describe in detail later in this chapter. Field names have a dotted syntax and appear on the left side of each comparison. The right side is a constant value that will be compared against the field. Here’s an example:

```
proc.name = emacs or proc.pid != 1234
```

Moving on:

> *Comparisons in filters are evaluated left to right, but parentheses can be used to define precedence.*

This means you can organize your filter using round brackets. For example:

```
proc.name = emacs or (proc.name = vi and container.name=redis)
```

Again, this works exactly the same as using brackets inside a logical expression in your favorite programming language. Now the final sentence:

> *A filter is applied to an input event and returns a Boolean result indicating if the event matches the filter.*

When you specify a filter in a Falco rule, the filter is applied to every input event. For example, if you're using one of the kernel drivers, filters are applied to every system call. The filter evaluates the system call and returns a Boolean value: TRUE or FALSE. TRUE means that the event satisfies the filter (we say that the filter *matches* the event), while FALSE means that the filter rejects, or drops, the event. For example:

```
proc.name = emacs or proc.name = vi
```

This filter matches (returns TRUE for) every system call generated by processes called emacs or vi.

That's essentially all you need to know at the high level. Now let's dive into the details.

Filtering Syntax Reference

Falco's filters offer all the traditional relational operators that you are used to finding in the most common programming languages. In addition, they include some operators that are specific to the type of matching you do in Falco.

Relational Operators

Table 6-1 provides a reference of all of the available operators, including an example for each of them.

Table 6-1. Falco's relational operat

ι
o
r
s

Operator	Notes	Example
=, !=	General equality/inequality operators. They can be used with every field.	`proc.name = emacs`
<=, <, >=, >	Numeric comparison operators. You can only use them with numeric fields.	`evt.buflen > 100`
contains	Can be used with string fields only. Performs a case sensitive string search of the given constant inside the field value, and returns TRUE if the field value contains the constant.	`fd.filename contains passwd`
icontains	Like contains, but case insensitive.	`user.name icontains john`
startswith	Can be used with string fields only. Returns TRUE if the given constant matches the beginning of the field value.	`fd.directory startswith "/etc"`
endswith	Can be used with string fields only. Returns TRUE if the given constant matches the end of the field value.	`fd.filename endswidth ".key"`

in	Compares the field value to multiple constants and returns TRUE if one or more constants equal to the field value. This operator can be used with any fields, including numeric fields and string fields.	`proc.name in (vi, emacs)`
intersects	Returns TRUE when a field with multiple values includes at least one value that matches with the provided list of constants.	`ka.req.pod.volumes.hostpath intersects (/proc, /var/run/docker.sock)`

pmatch	Returns TRUE if one of the constants is a prefix of the field value.
	Note: pmatch can be used as an alternative to the *in* operator, and performs better with large sets of constants thanks to the fact that it’s implemented internally as a trie instead of multiple comparisons.
		`fd.name pmatch (/var/run, /etc, /lib, /usr/lib)`
		`When fd.name = /var/run/docker` succeeds because `/var/run` is a prefix of `/var/run/docker`
		`When fd.name = /boot` does not succeed because no constant is a prefix of `/boot`.
		`When fd.name = /var` does not succeed because no constant is a prefix of `/var`.

exists	Returns TRUE if the given field exists for the input event.	`evt.res exists`
glob	Matches the given string against the field value according to Unix shell wildcard patterns. For more details: `man 7 glob`	`fd.name glob '/home/*/.ssh/*'`

Logical Operators

The logical operators that you can use in Falco filters are straightforward and don’t include any surprises. Just in case, Table 6-2 lists them and provides examples.

Table 6-2. Falco's logical operato

r
s

Operator	Example
and	`proc.name = emacs and proc.cmdline contains myfile.txt`
or	`Proc.name = emacs or proc.name = vi`
not	`not proc.name = emacs`

Strings and Quoting

String constants can be specified with no quotes:

```
proc.name = emacs
```

Quotes can, however, be used to enclose strings that include spaces or special characters. Both single quotes and double quotes are accepted. For example:

```
proc.name = "my process" or proc.name = 'my process'
```

This means you can include quotes in strings:

```
evt.buffer contains '"'
```

Fields

As you can see, Falco filters are not very complicated. However, they are extremely flexible and powerful. This power comes from the fields you can use in filtering conditions. Falco gives you access to a variety of fields, each of which exposes a property of the input events that Falco captures. Since fields are so important, let's see how they work and how they are organized. Then we'll discuss which ones to use and when.

Argument Fields Versus Enrichment Fields

Fields expose properties of input events as typed values. A field, for example, can be a string, like the process name, or a number, like the process ID.

At the highest level, Falco offers two categories of fields. The first category includes the fields that are obtained by dissecting input events. System call arguments, like the file name for an `open` system call or the buffer argument for a `read` system call, are examples of such fields. You access these fields with the following syntax (where X is the name of the argument you want to access):

```
evt.arg.X
```

Where N is the position of the argument, it looks like this:

```
evt.arg[N]
```

For example:

```
evt.arg.name = /etc/passwd
evt.arg[1] = /etc/passwd
```

To find out which arguments a specific event type supports, Sysdig is your friend. The output line for an event in Sysdig will show you all of its arguments and their names.

The second category consists of fields that derive from the enrichment process that libsinsp carries on while capturing system calls and other events. If you need a refresh on how this enrichment works, go back to the *state engine* section of chapter 3. Falco exports many fields that expose the content of libsinsp's thread and FD tables, effectively *enriching* the events received from the drivers.

To understand how this works, let's take the `proc.cwd` field as an example. For each system call that Falco captures, this field contains the current working directory of the process that issued the system call. This is handy if we want to capture all of the system calls generated by processes that are currently running "inside" a specific directory:

```
proc.cwd = /tmp
```

The working directory of the process is not part of the system call, so exposing this field requires tracking the working directory of a process and attaching it to every system call that the process generates. This, in turn, involves four steps:

1. Collect the working directory when a process starts, and storing it in the process's entry in the thread table.
2. Keep track of when the process changes its working directory (by intercepting and parsing the `chdir` system call) and updating the thread table entry accordingly.
3. Resolve the thread ID of every system call to identify the corresponding thread table entry.
4. Return the thread table entry's cwd value.

Libsinsp does all of this, which means that the `proc.cwd` field is available for every system call, not only for the directory-related ones like `chdir`. It's impressive how much hard work Falco does to expose this field to you.

Enrichment-based filtering is very powerful because it allows you to filter system calls (and any other event) based on properties that are not in the system call itself, but are of great use for security policies. For example, the following filter allows you to capture the system calls that read from or write to /etc/passwd.

```
evt.is_io=true and fd.name=/etc/passwd
```

It works even if these system calls originally don't contain any information about the file name (they operate on file descriptors). The hundreds of enrichment-based fields available out of the box are the main reason why Falco is so powerful and versatile.

Mandatory Fields Versus Optional Fields

Some fields exist for every input event, and you will be guaranteed to find them independently from the event type or family. Examples of such fields are `evt.ts`, `evt.dir`, and `evt.type`.

Most fields, on the other hand, are optional and only present in some input event types. Typically, you don't have to worry about it, as fields that don't exist will just evaluate to FALSE without generating an error. For example, the following check will just evaluate to FALSE for all of the events that don't have an argument called `name`.

```
evt.arg.name contains /etc
```

However, sometimes you might want to explicitly check if a field exists. You do that by using the *exists* relational operator:

```
fd.filename exists
```

Field Types

Fields have types, which are used to validate values and ensure the syntactic correctness of filters. Take the following filter:

```
proc.pid = hello
```

Falco and Sysdig will reject this filter with the following error:

```
filter error at position 16: hello is not a valid number
```

This happens because the `proc.pid` field is of type INT64. The typing system also allows Falco to improve the rendering of some fields by understanding the meaning behind them. For example, `evt.arg.res` is of type ERRNO, which by default is a number. However, when possible, Falco will resolve it into an error code string (such as EAGAIN), which improves the readability and usability of the field.

When we looked at relational operators, we noted how some are very similar to the ones in programming languages, while others are unique to Falco filters. The same is true for field types. Table 6-3 lists the types you can encounter in Falco filter fields.

Table 6-3. Field types

Type	Description
INT8, INT16, INT32, INT64, UINT8, UINT16, UINT32, UINT64, DOUBLE	Numeric types, like in your favorite programming language.
CHARBUF	A printable buffer of characters.
BYTEBUF	A raw buffer of bytes not suitable for printing.

ERRNO	INT64 value that, when possible, is resolved to an error code.
FD	INT64 value that, when possible, is resolved to the value of the file descriptor. For example, for a file FD this gets resolved to the file name; for a socket it gets resolved to the TCP connection tuple.
PID	INT64 value that, when possible, is resolved to the process name.
FSPATH	A string containing a relative or absolute file system path.
SYSCALLID	A 16 bit system call ID. When possible, the value gets resolved to the system call name.
SIGTYPE	An 8 bit signal number that, when possible, gets resolved to the signal name (e.g. SIGCHLD).
RELTIME	A relative time, with nanosecond precision, rendered as a human relative time interval.
ABSTIME	An absolute time interval.
PORT	A TCP/UDP port. When possible, resolved to a protocol name.
L4PROTO	A 1 byte IP protocol type. When possible, resolved to a L4 protocol name (TCP, UDP).
BOOL	A Boolean value.
IPV4ADDR	An IPv4 address.
DYNAMIC	Field type can vary depending on the context. Used for generic fields like evt.rawarg.
FLAGS8, FLAGS16, FLAGS32	A flags word that, when possible, is converted into a readable string (e.g. `O_RDONLY|O_CLOEXEC`). The resolution into the string is dependent on the context, as events can register their own flag values. So, for example, flags for an *lseek* system call event will be converted into values like `SEEK_END`, `SEEK_CUR` and `SEEK_SET`, while *sockopt* flags will be converted into `SOL_SOCKET`, `SOL_TCP` and so on.
UID	A unix user ID, resolved to a user name when possible.
GID	A unix group ID, resolved to a user name when possible.
IPADDR	Either an IPv4 or IPv6 address.
IPNET	Either an IPv4 or IPv6 network.
MODE	a 32 bit bitmask to represent file modes

How do you find out the type of a field you want to use? The best way is to invoke Falco with the --list and -v options:

```
./falco --list -v
```

This will print the full list of fields, including type information for each entry in the list.

Using Fields and Filters

Now that you've learned about filters and fields, let's take a look at how you can use them in practice. We'll focus on Falco and sysdig.

Fields and Filters in Falco

Fields and filters are at the core of Falco rules. Fields are used to express rules' conditions. Fields are part of both conditions and outputs. To demonstrate how, we'll craft our own rule.

Let's say we would like Falco to notify us every time there is an attempt to change the flags of a file and make it executable by another user. When that happens, we would like to know the name of the file that was changed, the new mode of the file, and the name of the user who caused the trouble. We would also like to know if the mode change attempt was successful or not.

Here is the rule:

```
- rule: File Becoming Executable by Others
  desc: Attempt to make a file executable by other users
  condition: (evt.type=chmod or evt.type=fchmod or
evt.type=fchmodat) and evt.arg.mode contains S_IXOTH
  output: attempt to make a file executable by others
(file=%evt.arg.filename mode=%evt.arg.mode user=%user.name
failed=%evt.failed)
  priority: WARNING
```

The condition section is where the rule's filter is specified.

File modes, including the executable bit, are changed using the `chmod` system call, or one of its variants. Therefore, the first part of the filter just selects events that are of type `chmod`, `fchmod` or `fchmodat`:

```
evt.type=chmod or evt.type=fchmod or evt.type=fchmodat
```

Now that we have the right system calls, we want to only accept the subset of them that sets the "other" executable bit. Reading the `chmod` manual page reveals that the flag we need to check is `S_IXOTH`. We determine its presence by using the *contains* operator:

```
evt.arg.mode contains S_IXOTH
```

Combining the two pieces with an *and* gives us the full filter. Easy!

Now, let's focus our attention on the *output* section of the rule. This is where we tell Falco what to print on the screen when the rule's condition returns TRUE. You will notice that this is just a printf-like string that mixes regular text with fields, whose value will be resolved in the final message:

```
attempt to make a file executable by others
(file=%evt.arg.filename mode=%evt.arg.mode user=%user.name
failed=%evt.failed)
```

The only thing you need to remember is that you need to prefix field names in the output string with the % character, otherwise they will just be treated as part of the string.

Time for you to try this! Save the rule above in a file called ch6.yaml. After that, run this command line in a terminal:

```
$ sudo falco -r ch6.yaml
```

In another terminal, run these two commands:

```
$ echo test > test.txt
$ chmod o+x test.txt
```

This is the output you will get in the Falco terminal:

```
17:26:43.796934201: Warning attempt to make a file executable by
others
(file=/media/psf/Home/git/plugins/plugins/cloudtrail/libcloudtrai
l.so
mode=S_IXOTH|S_IWOTH|S_IROTH|S_IXGRP|S_IWGRP|S_IRGRP|S_IXUSR|S_IW
USR|S_IRUSR user=root failed=false)
```

Congratulations, you've just performed your very own Falco detection! Note how `evt.arg.mode` and `evt.failed` are rendered in a human-readable way, even if internally they are numbers. This shows you the power of the filter/fields type system.

Fields and Filters in sysdig

The *observing system calls* section of chapter 4 offers a good introduction to sysdig. If you need a refresher, go take a look at that section. Here we will look into specifically how filters and fields are used in sysdig. (If you need a Sysdig refresher, see Chapter 4, particularly the section "Observing system calls."

While Falco is based on the concepts of rules and of notifying the user when rules match, sysdig focuses on investigation, troubleshooting, and threat-hunting workflows. In sysdig, you use filters to *restrict the input*, and you (optionally) use fields formatting to *control the output*. The combination of the two provides a ton of flexibility during investigations.

Filters in sysdig are specified at the end of the command line:

```
$ sudo sysdig proc.name=echo
```

Output formatting is provided using the `-p` command-line flag, and uses the same printf-like syntax that we just described when talking about Falco outputs:

```
$ sudo sysdig -p"type:%evt.type proc:%proc.name" proc.name=echo
```

An important thing to keep in mind is that, when the `-p` flag is used, sysdig will only print an output line for the events in which *all* of the specified filters exist. For example:

```
$ sudo sysdig -p"%evt.res %proc.name"
```

This command will print a line only for the events that have *both* a return value and a process name–skipping, for example, all the system-call "enter" events. If you care about seeing all of the events, put a star at the beginning of the formatting string:

```
$ sudo sysdig -p"*%evt.res %proc.name"
```

When a field is missing, it will be rendered as `<NA>`.

When no formatting is specified with `-p`, sysdig displays input events in standard format, which conveniently includes all of the arguments and argument names, for every system call. Here's an example sysdig output line for an openat system call, with the system call arguments highlighted in bold for visibility:

```
4831 20:50:01.473556825 2 cat (865.865) < openat
fd=7(<f>/tmp/myfile.txt) dirfd=-100(AT_FDCWD)
name=/tmp/myfile.txt flags=1(O_RDONLY) mode=0 dev=4
```

Each of the arguments can be used in a filter with the *evt.arg* syntax:

```
$ sudo sysdig evt.arg.name=/tmp/myfile.txt
```

As a more advanced example, let's convert the shell_in_container rule we just created for Falco into a sysdig command line:

```
$ sudo sysdig -p"attempt to make a file executable by others
(file=%evt.arg.filename mode=%evt.arg.mode user=%user.name
failed=%evt.failed)" (evt.type=chmod or evt.type=fchmod or
evt.type=fchmodat) and evt.arg.mode contains S_IXOTH
```

This shows how easy and nice it is to use sysdig as a development tool when creating new rules.

Falco’s Most Useful Fields

Here is a curated list of some of the most important Falco fields, organized by class. You can use this list as a reference when writing filters. For a full list, including all plugin fields, use the following at the command line:

```
$ ./falco --list -v
```

General

The fields listed in Table 6-4 apply to every event and include general properties of an event.

Table 6-4. evtfilter class

Field name	Description
evt.num	event number.
evt.time	event timestamp as a time string that includes the nanosecond part.

evt.dir	event direction can be either '>' for enter events or '<' for exit events.
evt.type	The name of the event (e.g. ‘open').
evt.cpu	number of the CPU where this event happened.
evt.args	all the event arguments, aggregated into a single string.
evt.rawarg	one of the event arguments specified by name. E.g. ‘evt.rawarg.fd’.
evt.arg	one of the event arguments specified by name or by number. Some events (e.g. return codes or FDs) will be converted into a text representation when possible. E.g. ‘evt.arg.fd’ or ‘evt.arg[0]'.
evt.buffer	the binary data buffer for events that have one, like read(), recvfrom(), etc. Use this field in filters with ‘contains’ to search into I/O data buffers.
evt.buflen	the length of the binary data buffer for events that have one, like read(), recvfrom(), etc.
evt.res	event return value, as a string. If the event failed, the result is an error code string (e.g. ‘ENOENT'), otherwise the result is the string ‘SUCCESS’.
evt.rawres	event return value, as a number (e.g. -2). Useful for range comparisons.
evt.failed	‘true’ for events that returned an error status.

Process

This class contains all the information you need about processes and threads. The information in Table 6-5 comes mostly from the process table that libsinsp constructs in memory.

Table 6-5. proc filter class

Field name	Description
proc.pid	the id of the process generating the event.

proc.exe	the first command line argument (usually the executable name or a custom one).
proc.name	the name (excluding the path) of the executable generating the event.
proc.args	the arguments passed on the command line when starting the process generating the event.
proc.env	the environment variables of the process generating the event.
proc.cwd	the current working directory of the event.
proc.ppid	the pid of the parent of the process generating the event.
proc.pname	the name (excluding the path) of the parent of the process generating the event.
proc.pcmdline	the full command line (proc.name + proc.args) of the parent of the process generating the event.
proc.loginshellid	the pid of the oldest shell among the ancestors of the current process, if there is one. This field can be used to separate different user sessions, and is useful in conjunction with chisels like spy_user.
thread.tid	the id of the thread generating the event.
thread.vtid	the id of the thread generating the event as seen from its current PID namespace.
proc.vpid	the id of the process generating the event as seen from its current PID namespace.
proc.sid	the session id of the process generating the event.
proc.sname	the name of the current process's session leader. This is either the process with pid=proc.sid or the eldest ancestor that has the same sid as the current process.
proc.tty	The controlling terminal of the process. 0 for processes without a terminal.

File Descriptor

Table 6-6 lists file descriptors, which are at the base of I/O. Details about files and directories, network connections, pipes and other types of inter process communication can all be found in this class.

Table 6-6. fd filter class

Field name	Description
fd.num	the unique number identifying the file descriptor.
fd.typechar	type of FD as a single character. Can be ‘f’ for file, 4 for IPv4 socket, 6 for IPv6 socket, ‘u’ for unix socket, p for pipe, ‘e’ for eventfd, ’s’ for signalfd, ‘l’

	for eventpoll, ‘i’ for inotify, ‘o’ for unknown.
fd.name	FD full name. If the fd is a file, this field contains the full path. If the FD is a socket, this field contain the connection tuple.
fd.directory	If the fd is a file, the directory that contains it.
fd.filename	If the fd is a file, the filename without the path.
fd.ip	(FILTER ONLY) matches the ip address (client or server) of the fd.
fd.cip	client IP address.
fd.sip	server IP address.
fd.lip	local IP address.
fd.rip	remote IP address.
fd.port	(FILTER ONLY) matches the port (either client or server) of the fd.
fd.cport	for TCP/UDP FDs, the client port.
fd.sport	for TCP/UDP FDs, server port.
fd.lport	for TCP/UDP FDs, the local port.
fd.rport	for TCP/UDP FDs, the remote port.
fd.l4proto	the IP protocol of a socket. Can be ‘tcp', ‘udp', ‘icmp’ or ‘raw’.

User and Group

Table 6-7 lists user and group filter classes, which, predictably, include fields related to users and groups.

Table6-7.userandgroupfilterclass

e
s

Field name	Description
user.uid	user ID.
user.name	user name.
group.gid	group ID.
group.name	group name.

Container

The container class (Table 6-8) can be used for everything related to containers, including obtaining IDs, names, labels and mounts.

Table 6-8.container filter class

e
s

Field name	Description
container.id	the container id.
container.name	the container name.
container.image	the container image name (e.g. falcosecurity/falco:latest for docker).
container.image.id	the container image id (e.g. 6f7e2741b66b).
container.privileged	true for containers running as privileged, false otherwise.
container.mounts	A space-separated list of mount information. Each item in the list has the format <source>:<dest>:<mode>:<rdrw>:<propagation>.
container.mount	Information about a single mount, specified by number (e.g. container.mount[0]) or mount source (container.mount[/usr/local]). The pathname can be a glob (container.mount[/usr/local/*]), in which case the first matching mount will be returned. The information has the format <source>:<dest>:<mode>:<rdrw>:<propagation>. If there is no mount with the specified index or matching the provided source, returns the string "none" instead of a NULL value.
container.image.repository	the container image repository (e.g. falcosecurity/falco).
container.image.tag	the container image tag (e.g. stable, latest).
container.image.digest	the container image registry digest (e.g. sha256:d977378f890d445c15e51795296e4e5062f109ce6da83e0a355fc4ad8699d27).

Kubernetes

When Falco is configured to interface with the Kubernetes API server, this class (listed in Table 6-9) can be used to fetch information about Kubernetes objects.

Table 6-9. k8s filter classes

Field name	Description

k8s.pod.name	Kubernetes pod name.
k8s.pod.id	Kubernetes pod id.
k8s.pod.label	Kubernetes pod label. E.g. ‘k8s.pod.label.foo’.
k8s.rc.name	Kubernetes replication controller name.
k8s.rc.id	Kubernetes replication controller id.
k8s.rc.label	Kubernetes replication controller label. E.g. ‘k8s.rc.label.foo’.
k8s.svc.name	Kubernetes service name (can return more than one value, concatenated).
k8s.svc.id	Kubernetes service id (can return more than one value, concatenated).
k8s.svc.label	Kubernetes service label. E.g. ‘k8s.svc.label.foo’ (can return more than one value, concatenated).
k8s.ns.name	Kubernetes namespace name.
k8s.ns.id	Kubernetes namespace id.
k8s.ns.label	Kubernetes namespace label. E.g. ‘k8s.ns.label.foo’.
k8s.rs.name	Kubernetes replica set name.
k8s.rs.id	Kubernetes replica set id.
k8s.rs.label	Kubernetes replica set label. E.g. ‘k8s.rs.label.foo’.
k8s.deployment.name	Kubernetes deployment name.
k8s.deployment.id	Kubernetes deployment id.
k8s.deployment.label	Kubernetes deployment label. E.g. ‘k8s.rs.label.foo’.

Cloudtrail

Cloudtrail fields (listed in Table 6-10), are available when the Cloudtrail plugin is configured. They allow you to build filters and formatters for AWS detections.

Table 6-10. cloudtrail filter class

e
s

Field name	Description
ct.error	The error code from the event. Will be “” if there was no error.
ct.src	the source of the cloudtrail event (eventSource in the json).
ct.shortsrc	the source of the cloudtrail event (eventSource in the json, without the ‘.amazonaws.com’ trailer).
ct.name	the name of the cloudtrail event (eventName in the json).
ct.user	the user of the cloudtrail event (userIdentity.userName in the json).
ct.region	the region of the cloudtrail event (awsRegion in the json).
ct.srcip	the IP address generating the event (sourceIPAddress in the json).
ct.useragent	the user agent generating the event (userAgent in the json).
ct.readonly	‘true’ if the event only reads information (e.g. DescribeInstances), ‘false’ if the event modifies the state (e.g. RunInstances, CreateLoadBalancer...).
s3.uri	the s3 URI (s3://<bucket>/<key>).
s3.bucket	the bucket name for s3 events.
s3.key	the S3 key name.
ec2.name	the name of the ec2 instances, typically stored in the instance tags.

Conclusion

Congratulations, you are now a filtering expert! At this point, you should be able to read and understand Falco rules, and you are much closer to being able to write your own. In the next chapter, we will devote our attention to Falco outputs.

Chapter 7. The Output Framework

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the seventh chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

In previous chapters, you learned how Falco collects events (its input) and how it treats them to allow you to receive important security notifications (its output). At the end of this processing pipeline, a key piece of Falco enables it to deliver notifications (also called *alerts*) to the right place: the *output framework*.

We call it a *framework* because its modular design provides all you need to deliver notifications to any destination you wish. In this chapter, you will learn how the output framework works and how you can configure and extend it.

Falco Output Architecture

The *output framework* is the last piece of the event-processing pipeline that we have been describing in this part of the book. Falco's userspace program implements the core mechanism internally, but external tools can extend it. Its job is to deliver *notifications* (also called *alerts*) to the correct destination timely. Whenever an upstream event (produced by a driver, a plugin, an internal, or any other input source supported by Falco) meets a rule's condition, the rule engine asks the output framework to send out a notification. This comes into play when Falco needs to send a notification to a downstream consumer, which may be any other program or system in your environment (or simply you).

The process of delivering alerts involves two distinct stages, as pictured in Figure 7-1.

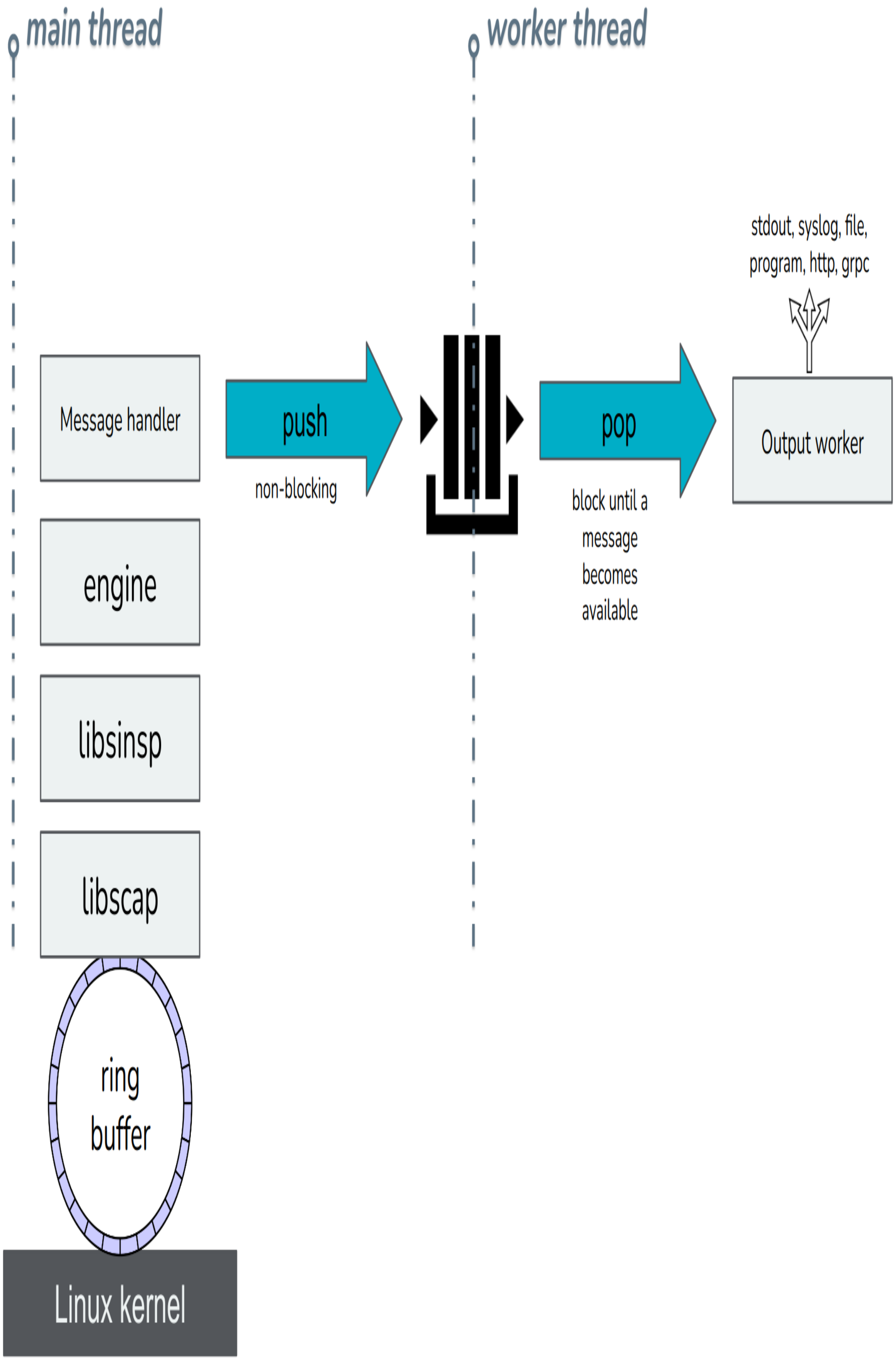

main thread
worker thread
stdout, syslog, file,
program, http, grpc
Message handler
push
non-blocking
pop
Output worker
block until a
message
becomes
available
engine
libsinsp
libscap
ring
buffer
Linux kernel

Figure 7-1. The two stages of delivering notifications

In the first stage, a *handler* receives the event data and information about the event-triggered rule. It prepares the notification using the provided information and formats the textual notification representation according to the rule's `output` field. Then, to avoid an output destination blocking the processing pipeline (which runs in the main execution thread), the handler pushes the notification into a concurrent queue.[1] The push operation is non-blocking, so the processing pipeline does not need to wait for the notification consumer to pull the notification; it can continue to do its job without interruption. Indeed, Falco needs to perform this stage as quickly as possible so that the processing pipeline can process the next event.

On the other end of the queue, the *output worker* (which runs in a separate execution thread) is waiting to pop notifications from the queue. This is when the second stage begins. Once the output worker receives a notification, it immediately fans the notification out to all configured *output channels*. An *output channel* (or simply an *output*) is a part of the output framework that allows Falco to forward alerts to a destination. Each output channel implements the actual logic to notify a particular class of alert consumers. For instance, some consumers want notifications written to a file, while others prefer them to be posted to a web endpoint. (See the "Output channels" section later in this chapter.)

This two-stage approach allows the processing pipeline to run without interference from the output delivery process. However, something can still go wrong in the delivery. In particular, when delivering a notification involves I/O operations, those may block temporarily (for example, a network slowdown) or indefinitely (for instance, when there's no space left on the disk). The queue in the middle of the two stages works well in absorbing temporary slowdowns so well that you won't even notice them. (By default, Falco can accumulate pending notifications in the queue for up to two seconds.)

When the recipient of a notification blocks for a long time (or indefinitely), there's nothing that Falco can automatically do to address such a situation.

As a last resort, it will try to inform you about what happened by logging into the standard error stream (*stderr*). When this happens, it is usually a symptom of misconfigurations (for example, the path to the destination is wrong) or insufficient resources (no space left in the destination), which the user is asked to manually fix.

Once the notification delivery process completes, Falco's userspace program has accomplished its purpose. It's then up to the notification consumer to decide what to do with the alert.

You have learned that the design of the output framework takes care of possible issues and accommodates many different use cases. It is also flexible enough to allow you to receive notifications in many ways and at many destinations. The rest of this chapter will give you details about all the available possibilities. We'll also take a quick look at some other tools that allow you to further extend output processing before delivering the notification to the final destination. (We go deeper into this in Chapter 13.)

Output Formatting

Output formatting happens in the first stage of notification delivery. It applies formatting to the notification before forwarding it to the output channel and allows you to customize how Falco presents notifications to its consumers so that you can easily integrate them with your specific use case.

Two options in the Falco configuration (*/etc/falco/falco.yaml*) control this operation. The first controls the formatting of the timestamp:

```
time_format_iso_8601: false
```

If the option is `false` (the default value), Falco will display dates and times according to the */etc/localtime* settings. If `true` (the default value when Falco is running in a container), Falco will use the ISO8601 standard for representing dates and times. Note that this option controls not only output notifications but also any other message that Falco logs.

The second option is actually a set of options that enable JSON formatting for the notification. By default, JSON formatting is disabled:

```
json_output: false
```

If `false`, Falco formats the notification as a plain text string (including the timestamp, the severity, and the message). If `true`, Falco encloses the notification in a JSON-formatted string, including several fields. The following two options allow you to include or exclude some of those fields from the output:

```
json_include_output_property: true
```

If enabled (the default), you will still find the plain text representation of the notification in the `output` of the JSON object. Just disable this option to save a few bytes if you don't need it.

```
json_include_tags_property: true
```

If enabled, you will find the `tags` field in the JSON object. It will contain an array of tags as specified in the matching rule. Rules with no tags defined will get an empty array (`tags: []`) in the output. If you disable this option, you won't get the `tags` field in the JSON object.

> **NOTE**
>
> `json_output` is not an output channel, despite its name. The `json_output` configuration only controls the notification formatting applied upstream to output channels and thus may affect their final output content. Only the channels described in the next section are output channels.

Output Channels

Falco comes with six built-in output channels, as shown in Table 7-1. We will describe each of them in the following subsections. By default, only two channels are enabled (the *stdout output* and the *syslog output*). However, Falco allows you to simultaneously enable as many channels as you need.

Table 7-1. Falco’s built-in output

channels

Channel	Description
Standard output	Send notifications to the Falco’s standard output (i.e., *stdout*)
Syslog output	Send notifications to the system via Syslog
File output	Write notifications to a file
Program output	Pipe notifications to a program’s standard input
HTTP output	Post notification to an URL
gRPC output	Allow a client program to consume notification via a gRPC API

You can find all the options to configure those outputs in the Falco configuration file (*/etc/falco/falco.yaml*). Note that all the configuration snippets present in this section and subsections refer to the Falco configuration file.

Each output channel has at least one option called `enabled`, which can be `true` or `false`. Other options may be available only for specific output (you will discover them soon).

Furthermore, some global options can affect the functioning of all or some output channels. One option (which you saw in the previous section) is `json_output`. When enabled, the alert’s message text will be JSON formatted, regardless of the output channel used. The other global options that can affect the output channels behavior are listed in Table 7-2.

Table 7-2. Global options for outpu

tchannels

Global option (with default)	Description
`buffered_outputs: false`	Enable and disable full buffering in output channels. When disabled, Falco flushes to the output of every alert immediately, which may generate higher CPU usage but is useful when piping outputs into another process or a script. Unless you encounter an issue with the default value, you usually don’t need to enable this option. Note that Falco’s command line flag —`-unbuffered` can override this option. Not all output channels observe this global option. Some output channels may implement specific buffering strategies that you cannot disable.
`output_timeout: 2000`	The value specifies the duration (in milliseconds) to wait before considering the delivery notification deadline exceed. When the notification consumer blocks and the output channel cannot deliver an alert within a given deadline, Falco reports an error indicating which output is blocking the notifications. A such as error indicates a misconfiguration issue or I/O problem in the consumer that Falco cannot recover.

`outputs:`
`rate: 1`
`max_burst: 1000`
	These options control the notification rate limiter so that output channels do not flood their destinations.
	The rate limiter implements a token bucket algorithm. The `rate` sets the number of tokens (such as right to send a notification) gained per second, `max_burst` sets the maximum number of tokens outstanding.
	With default values, Falco can send up to 1,000 notifications in a row; then it caps the notification rate to 1 notification per second. When there’s no notifications activity, Falco gains the rate limit by 1 each second until it restores back the full burst.

TIP

Although not strictly related to the output mechanism, other Falco settings may affect what you will receive in the output. For example, the configuration `priority: debug` controls the minimum rule priority level to load and run. Also, the command-line option `-t <tag>` allows you to load only those rules with a specific tag. In those cases, clearly, you won't get any output regarding rules that Falco does not load. In general, you should assume that any rules-related option or configuration could indirectly affect the output.

Now that you've learned what channels are and what settings can change their behavior, let's go through each output channel.

Standard Output

The *standard output* (`stdout_output` in the configuration file) is the most straightforward channel implemented by Falco and is enabled by default. When enabled, Falco will print a line for each alert. It allows you to see alert notifications when manually running Falco from a console or when looking at a container or a Kubernetes Pod log. The only option specifically available for this channel is `enabled` (which can be either `true` or `false`). However, this output channel is also affected by the global buffering option (`buffered_outputs`). When the outputs are buffered, the *stdout* stream will be fully buffered or line-buffered if the stream is an interactive device (such as in a TTY).

Syslog Output

The *syslog output* (`syslog_output` in the configuration file) allows Falco to send a *Syslog* message for each alert. This is the only other output channel enabled by default (along with standard output). The only option specifically available for this channel is `enabled` (which can be either true or false). When enabled, Falco sends messages to Syslog with a facility of LOG_USER[2] and a severity level equal to the priority value defined by the rule.

Depending on the Syslog daemon you are using, you can read those messages using commands like tail -f /var/log/syslog or journalctl -xe. The actual message format depends on the Syslog daemon, too.

File Output

If you enable the *file output*, Falco will write each alert to a file. The default configuration for this output channel is:

```
file_output:
  enabled: false
  keep_alive: false
  filename: ./events.txt
```

The filename option allows you to specify the destination on the file on which Falco will write. It will also create the file if it does not yet exist and will not try to truncate or rotate the file if it exists already.

With keep_alive disabled (the default), Falco will open the file for appending, write the message, and then close the file for each alert. Otherwise, if keep_alive is set to true, Falco will only open the file once before the first alert and keep it open for all subsequent alerts.

Whether keep_alive is enabled or not, Falco closes and reopens the file when it receives a SIGUSR1 signal. This feature is handy if you'd like to use a program to rotate the output file (for example, logrotate).

Finally, writing to a file is generally buffered unless you disabled the global buffering option. Closing the file will flush the buffer.

Program Output

The *program output* is very similar to the file output, but, in this case, Falco will write the content of each alert to the standard input of a program you specify in the configuration file. The default configuration for this output channel is:

```
program_output:
  enabled: false
  keep_alive: false
  program: "jq '{text: .output}' | curl -d @- -X POST
https://hooks.slack.com/services/XXX"
```

The `program` field allows you to specify the program that will receive alerts in its standard input. Falco runs the program via a shell, so you can specify a command pipeline if you wish to add any processing step before delivering the message to the final program. The `program`’s default value shows you a nice example of its usage: when executed, that one-liner posts the alert to a Slack Webhook endpoint. (However, using Falcosidekick would be a better option; see Chapter 12).

If `keep_alive` is set to `false`, Falco re-spawns the program and writes its standard input on each alert to deliver. If `keep_alive` is set to `true`, Falco starts the program once (right before the first alert) and keeps the program pipe open for delivering subsequent alerts.

Falco closes and reopens the program when it receives a `SIGUSR1` signal. However, note that the program runs in the same process group as Falco; thus, the program gets all signals that Falco receives. It’s up to you to override the program signal handler if you need to.

Buffering is supported via the global option. When Falco closes the program, it also flushes the buffer.

HTTP Output

When you need to send alerts over an HTTP(s) connection, the best choice is to use the *http output*. Its default configuration is straightforward:

```
http_output:
  enabled: false
  url: http://some.url
```

Once enabled, the only other configuration you need to specify is the `url` of your endpoint.

Falco will make an HTTP POST request to the specified URL on each alert.

Both unencrypted HTTP and secure HTTPS endpoints are supported. Buffering for this output channel is always enabled (even if you disable the global buffering option).

Furthermore, this output channel is preferred when you use Falcosidekick, a simple daemon for connecting Falco to your ecosystem. It takes Falco's alerts and forwards them fan-out style to many different destinations (more than 50 are available at the time of writing).

If you want Falco to forward alerts to Falcosidekick, apply this Falco configuration:

```
json_output: true
json_include_output_property: true
http_output:
  enabled: true
  url: "http://localhost:2801/"
```

Note the above configuration assumes you already have Falcosidekick running and configured to listen to `localhost:2801`. Change it accordingly if your setup is different. You can find all the details about configuring Falcosidekick in Chapter 13 and in its online documentation.

gRPC Output

The *gRPC output* is likely the most sophisticated output channel. Unlike the other outputs, it allows greater control over alert forwarding and full granularity in the information received. This output channel is for you if you'd like to send alerts to an external program connected via Falco's gRCP API. Its default configuration is:

```
grpc_output:
  enabled: false
```

As you can see, it’s disabled by default.

However, before you enable it, there’s something to consider. Falco comes with a gRPC server that exposes the API. You will need to enable both the gRPC server and the gRPC output (we will see how to do that in a moment). The API provides several gRPC services, only some of which are related to the gRPC output. One service allows you to get all outputs present in the system up to the service call. Another allows subscribing to a stream of outputs. Client programs can decide which implementations best fit their needs. In both cases, when the gRPC output is enabled, Falco uses an internal queue to temporarily store alerts until the client program consumes them. This means you should not enable the gRPC output if there’s no client program that consumes the alerts; otherwise, the internal queue may grow indefinitely. The global buffering option does not affect this output channel.

With that in mind, to make this output channel work, the first thing you have to do is to enable the gRPC server. It supports two binding types: over a Unix socket and over the network with mandatory mutual TLS authentication.

First, let’s see the gRPC server over a Unix socket:

```
grpc:
  enabled: true
  bind_address: "unix:///var/run/falco.sock"
  threadiness: 0
```

And here is the gRPC server over the network with mandatory mutual TLS authentication:

```
grpc:
  enabled: true
  bind_address: "0.0.0.0:5060"
  threadiness: 0
  private_key: "/etc/falco/certs/server.key"
  cert_chain: "/etc/falco/certs/server.crt"
  root_certs: "/etc/falco/certs/ca.crt
```

Both binding types offer the same gRPC functionalities, so you can choose the one that satisfies your needs. Once you have enabled the gRPC server, the next step is to enable the gRCP output:

```
grpc_output:
  enabled: true
```

Finally, you will have to configure your client program to connect to the Falco gRPC API. This mostly depends on the program you are using.

Notably, two programs can connect to this output (see Chapter 2). One is falco-exporter, which connects to the Falco gRPC API to export metrics consumable by Prometheus (more details in Chapter 13). Another is the event-generator, which can optionally connect to the Falco gRPC API to test if fake events are actually processed (helpful when developing integration tests). You can also implement your own program. The Falcosecurity organization provides SDK, which easily allows you to create gRPC client programs for Falco in several programming languages (for example, client-go for Golang, client-rs for Rust, and client-py for Python). You can find more information about developing with the Falco gRPC API in Chapter 14.

Last but not least, here is an extract from the proto-definition of the message that Falco sends via the gRCP API:

```
// The `response` message is the representation of the output
model.
// It contains all the elements that Falco emits in an output
along
// with the definitions for priorities and source.
message response {
  google.protobuf.Timestamp time = 1;
  falco.schema.priority priority = 2;
  falco.schema.source source = 3;
  string rule = 4;
  string output = 5;
  map<string, string> output_fields = 6;
```

```
    string hostname = 7;
    repeated string tags = 8;
  }
```

The `response` message includes the already-formatted alert string (that you will find in the `output` field) as well as all pieces of information, split across various fields. The client program can assemble and process them in any way it needs. That's very useful if you want to build your own application on top of Falco.

Other Logging Options

So far we've described the core part of the output framework. Now let's look at a few options to help you in troubleshooting. Like most applications, Falco can output debugging information and errors. Those informative messages are about the functioning of Falco itself and are not its primary output.

Falco internally implements various logging messages. They can vary from one release to another. A common example of this logging is the initial information that Falco prints out when it starts. Another less common case is when Falco informs you that it was not able to load the driver:

```
Mon Dec 20 14:00:23 2021: Unable to load the driver.
```

> **NOTE**
>
> Logging does *not* refer to the process of output security notifications. The log messages discussed in this section are not security notifications. Logging options do not affect notification processing in any way. Also, since those logs are not notifications, Falco does not output them through the output channels. Although you might see the usual notifications interleaved with logging messages when running Falco in a terminal, keep in mind that they are different.

Falco outputs those messages via the standard error stream and sends them to Syslog. You can configure Falco to discard some messages based on a severity level. Table 7-3 lists the logging options you can configure from Falco’s configuration file (*/etc/falco/falco.yaml*).

Table 7-3. Options for Falco's int

ernal logging

Logging option (with default)	Description
`log_stderr: true`	If enabled, Falco sends log messages to the standard error stream (i.e. *stderr*)
`log_syslog: true`	If enabled, Falco sends log messages to Syslog. Note that this option is not related to the Syslog output and does not affect it.
`log_level: info`	This option defines the minimum log level to include in logs. Values can be: `emergency`, `alert`, `critical`, `error`, `warning`, `notice`, `info`, or `debug`. Note that those values, although similar, are not rule priority levels.

Conclusion

This chapter concludes Part II of this book. You should now be familiar with the Falco architecture and its inner workings. You've learned the processing pipeline’s data flow, ending with the output framework, which will allow you to use Falco in a variety of ways. For example, you can view security notifications on your favorite dashboard or even create a response engine (a mechanism that takes action when a specific event occurs) on top of Falco. To discover all possible use cases, use your imagination—and continue reading this book.

The next level up is real-world use cases, so the next part of the book is all about running Falco in production. Do you feel ready? As always, we will guide you through each step.

[1] A *concurrent queue* is a way of implementing a queue data structure that multiple running threads can safely access in parallel. The *pop* and *push* operations are typical actions that a queue supports (respectively, to enqueue and dequeue an item). Most implementations allow performing those operations in either blocking or non-blocking fashions.

[2] In the Syslog protocol, the facility value determines the function of the process that created the message. LOG_USER is intended for messages generated by user-level applications.

Chapter 8. Installing Falco

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the eighth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

Welcome to Part III of this book, which will walk you through using Falco in the real world. You have already learned how Falco and its architecture work, so the next level up for you is to start protecting your production applications and systems.

In this chapter, you will find what you need to know to install Falco in production. We will show you different scenarios and common best practices so that you can find the right instructions for your use case.

We will start by giving you an overview of common scenarios. Then we will describe several installation methods for each of them. We strongly recommend reading about all of the installation methods, even if you need only some of them, to get a complete picture of the possibilities and choose which fits your needs best.

Choosing Your Setup

The Falco Project officially supports three ways to run Falco in production:

- Running Falco directly on a host
- Running Falco in a container
- Deploying Falco to a Kubernetes cluster

Each option has a different installation method. However, there are a few differences between the first option and the others. Installing Falco directly on the host is your *only* choice when your environment does not come with a container runtime or Kubernetes. It is also the most secure way to run Falco, because it is isolated from the container system (and thus difficult to breach in case of compromise). Even so, installing Falco directly on the host is usually the most difficult of these options to maintain. It’s also not always possible (for example, when your applications live in a managed Kubernetes cluster and you don’t have full access to the host machines). The other options are usually more straightforward and easier to manage. Especially if your applications run on a Kubernetes cluster, deploying Falco to Kubernetes is a common choice. Consider the pros and cons and your requirements before making your choice.

Before installing Falco with any of these methods, you need to choose which scenario Falco will run, which can change the installation process and configuration significantly. The two most common scenarios are monitoring syscalls and working with other data sources.

The default scenario is instrumenting the system to monitor *syscalls*. You will need to deploy a Falco sensor on each machine or cluster node you need to monitor, as well as install a driver on each underlying host.

When you work with *other data sources***,** like those provided by plugins, you will likely need to install only one Falco sensor (or one for each event producer), and you won’t need a driver. Although each data source may have small differences, we can treat them as a single installation scenario because the installation process is very similar. Generally, this latter scenario has fewer requirements and is simpler to deploy or install.

Multiple scenarios can apply simultaneously. If you need to satisfy more than one scenario at the same time or your scenario requires multiple Falco sensors, you will need one Falco installation per scenario. You can then aggregate the notifications coming from each sensor by using other tools, like Falcosidekick (see Chapter 12).

Your final setup will depend on your needs and choices. The following sections provide instructions for each installation method in the two scenarios mentioned above.

Installing Directly on the Host

Installing Falco directly on the host is a straightforward task. You learned the essential aspects in Chapter 2. This installation method is mainly intended for the default scenario that uses system calls to secure and monitor a system. For that reason, this installation method also installs the driver and configures Falco to use it. Chapter 9 discusses how to change the Falco configuration and set it up for other data sources.

This method installs the following:

- The userspace program `falco`
- The driver (Kernel module by default)
- The default configuration file and the default ruleset files in */etc/falco*
- The `falco-driver-loader` utility (you may use to manage the driver)
- Few bundled plugins (may vary from version to version)

To install Falco, you will use one of the following artifacts provided by Falco’s download page:

- RPM package
- DEB package
- Binary package

You should use one of the first two packages if you intend to install Falco via a compatible package manager; otherwise, use the binary package. Read on for more details.

> **NOTE**
>
> The following subsections include various commands that you need to run on your system. You need to execute almost all of them with enough privileges (for example, using `sudo`).

Using a Package Manager

This installation method is for Linux distributions with a package manager that supports *deb* or *rpm* packages.

The setup process for a `.deb` or `.rpm` package will also install a systemd unit to use Falco as a service on your system, as well as the Kernel module (the default driver) via dkms.

APT and *YUM* are the most popular package managers that allow installing, respectively, deb and rpm packages. (If your package manager is different but still supports deb or rpm packages, the installation procedure will be very similar, though the exact instructions may vary. Refer to its documentation.)

Using APT (.deb package)

APT is the default package manager for Debian and Debian-based distributions like Ubuntu. It allows you to install software applications distributed as `.deb` packages. To install Falco using apt, you first need to trust the Falco Project's GPG key and configure the apt repository that holds Falco packages:

```
curl -s https://falco.org/repo/falcosecurity-3672BA8F.asc | apt-
key add -

echo "deb https://download.falco.org/packages/deb stable main" |
tee -a /etc/apt/sources.list.d/falcosecurity.list
```

Then update the apt package list:

```
apt-get update -y
```

Since this installation method will also install Falco’s Kernel Module, you must install the Linux Kernel headers as a precondition:

```
apt-get -y install linux-headers-$(uname -r)
```

Finally, install Falco:

```
apt-get install -y falco
```

Using *yum* (.rpm package)

YUM is a command-line utility for Linux distributions that use the RPM Package Manager, such as CentOS, RHEL, Fedora, and Amazon Linux. It allows you to install software applications distributed as .rpm packages. Before installing Falco with yum, you must ensure that the `make` package and the `dkms` package are present in your system. You can check that by running:

```
yum list make dkms
```

If they are not present, install them:

```
yum install epel-release
yum install make dkms
```

Next, trust the Falco Project’s GPG key and configure the RPM repository that holds Falco packages:

```
rpm --import https://falco.org/repo/falcosecurity-3672BA8F.asc

curl -s -o /etc/yum.repos.d/falcosecurity.repo
https://falco.org/repo/falcosecurity-rpm.repo
```

Since this installation method will also install Falco’s Kernel Module, you must install the Linux Kernel headers as a precondition:

```
yum -y install kernel-devel-$(uname -r)
```

TIP

If `yum -y install kernel-devel-$(uname -r)` does not find the kernel headers package, run `yum distro-sync` and then reboot the system. After the reboot, try the above command again.

Finally, install Falco:

```
yum -y install falco
```

Complete the installation

You should now have the Kernel Module installed via dkms and a systemd unit installed to run Falco as a service.

Before you start using Falco, you need to enable the Falco systemd service:

```
systemctl enable falco
```

Your installation is complete. The service will automatically start running at the next reboot. If you want to start it immediately, just run:

```
systemctl start falco
```

From now on, you can manage the Falco service through the function provided by systemd.

Switch to eBPF

Falco packages use the Kernel Module by default. It is usually the best choice when installing Falco directly on the host. However, if you have particular requirements or other reasons not to use the Kernel Module, you can easily switch to the eBPF probe.

First, make sure you have an eBPF probe installed in your system. You can install it using the falco-driver-loader script, as explained below in the “Managing the driver” subsection.

Then, you need the systemd unit file, located in */usr/lib/systemd/user/falco.service*. You can use `systemctl edit falco` to modify it. You need to add an option to set the `FALCO_BPF_PROBE` environment variable in that file. Also, in the same section, comment (or remove) the `ExecStartPre` and `ExecStartPost` options under the `[Service]` section, so the Falco service will not load the Kernel Module anymore. The final content of the *falco.service* file should look like the following excerpt:

```
[Unit]
Description=Falco: Container Native Runtime Security
Documentation=https://falco.org/docs/
[Service]
Type=simple
User=root
Environment='FALCO_BPF_PROBE=""'
#ExecStartPre=/sbin/modprobe falco
ExecStart=/usr/bin/falco --pidfile=/var/run/falco.pid
#ExecStopPost=/sbin/rmmod falco
```

Once done, don’t forget to restart the Falco service:

```
systemctl restart falco
```

Falco should now start using the eBPF probe.

Using a plugin

Falco packages come configured with the syscalls instrumentation scenario, so the included systemd unit loads the driver when Falco starts. However, when using a plugin, you don’t need to load the driver. To leave that out, edit the */usr/lib/systemd/user/falco.service* and remove (or comment) the

ExecStartPre and ExecStartPost options. Optionally, you can also configure the service to run Falco with a less privileged user by modifying the value of the User option.

Next, you'll need to configure Falco to use the plugin of your choice, which we'll explain in chapter 9. Then you'll just need to restart the Falco service. Falco will run using the new configuration.

Without Using a Package Manager

Installing Falco without using a package manager is extremely quick. This installation method is intended for distributions that do not support a compatible package manager. We explained all of these steps in detail in Chapter 2, but here are shortened instructions.

All you need to do is grab the link to the latest available version of the binary package from the Falco's download page, then download it in a local folder:

```
curl -L -O https://download.falco.org/packages/bin/x86_64/falco-
0.30.0-x86_64.tar.gz
```

Then extract the package and copy its content to your filesystem's root:

```
tar -xvf falco-0.29.0-x86_64.tar.gz
cp -R falco-0.29.0-x86_64/* /
```

Finally, install the driver manually before using Falco. (You will find the instructions in the following subsection.) You don't need to install the driver if you want to use a plugin. Also, note that the binary package does not provide a systemd unit nor any other mechanism to run Falco when your system starts automatically, so whether to execute Falco or run it as a service is entirely up to you.

Managing the Driver

Unless you are not interested in using syscalls as a data source, you will likely need to manage the driver. If you installed Falco without a package manager, install the driver before using Falco manually. All packages provide a helpful script called `falco-driver-loader` (see Chapter 2). If you followed the instructions earlier in this chapter, you should already have it installed in your system.

Our suggestion is to familiarize yourself with the script by using –help to get its command-line usage. To do that, just run:

```
falco-driver-loader --help
```

This lets you do several actions, including installing the driver (either the Kernel Module or the eBPF probe) by compiling it or downloading it. It also allows you to remove a previously installed driver.

You can also run the script without any option:

```
falco-driver-loader
```

If you do, by default, it will try to install a Kernel Module via dkms. To be precise, it will first try to download a prebuilt driver, if any. Otherwise, it will try to compile the driver locally. The script will also inform you if any required dependencies are missing (for example, if `dkms` or `make` are not present in your system).

If you want to install the eBPF probe instead, run:

```
falco-driver-loader bpf
```

Running Falco in a Container

The Falco project provides several container images that you can use to run Falco in a container. Although the Falco container images described in this section work with almost any container runtime, we use Docker in our examples for simplicity. If you want to use a different tool, you can apply the same concepts.

You can also use the Falco container images to run Falco in Kubernetes. Even if you are only interested in deploying Falco on Kubernetes, we still advise you to read this section. It will give you essential concepts.

Table 8-1 lists the main available images, and you can get them from Falco's download page. These images contain all the necessary components to install the driver and run Falco. Further into the section, we'll discuss how to use them to support some common use cases.

Table 8-1. Falco containerimagesho

sted by the docker.io registry

Image name	Description

`falcosecurity/falco`	Default Falco image. It contains Falco, the falco-driver-loader script, and the building toolchain (required to build the driver on the fly). The entry point of this image will call the falco-driver-loader script to automatically install the driver on the host before running Falco in the container.
`falcosecurity/falco-driver-loader`	This image is similar to the default one, but it will not run Falco. The image entry point will only run the falco-driver-loader script. You can use it when you want to install the driver at a different moment or when using the Principle of Least Privilege (more about that later in this chapter). Since this image alone cannot run Falco, use it in combination with another image, like `falcosecurity/falco-no-driver`.
`falcosecurity/falco-no-driver`	This alternative to the default image only contains Falco, so it cannot install the driver. Use it when using the Principle of Least Privilege or when your data source does not need a driver (for example, when using a plugin as a data source).

Different tags are available for each distributed image. Tags allow you to choose a specific Falco version: for example, `falcosecurity/falco:0.31.0` contains Falco's 0.31.0 release. Usually, you'll just use *latest*, which points to the latest released version of Falco.

If you want to experiment with a not-yet-released version of Falco, the *master* tag ships the latest available development version. An automatic process builds and publishes images with this tag every time new code changes are merged into the master branch of Falco's GitHub repository. This means it is not a stable release–don't use it in production unless you want to try an experimental feature or debug a particular issue. Generally, we suggest always using the `latest` tag, since it ships the latest Falco version and ruleset updates.

Next, we will describe how to use those images in the same two common scenarios we've been using: syscalls instrumentation, which requires a driver, and using a plugin as a data source, which does not.

Syscalls Instrumentation Scenario

A Falco driver (either a Kernel Module or an eBPF probe) installed directly on the host is required for syscalls instrumentation. Falco needs to run with enough privileges to interact with the driver; of course, if you want to use a container image to install the driver, that image needs to run with full privileges.

The Falco Project provides two modes of installing the driver on the fly and then running Falco in a container. The first and simplest mode uses just one container image with full privileges. The second uses two images: one image that temporarily runs with full privileges just to install the driver, and another image that then runs Falco with lesser privileges. The second approach allows enhanced security since the long-running container gets a restricted set of privileges, making life harder for a possible attacker. We recommend using least privileged mode to run Falco in a container.

Fully privileged mode

Running Falco in Docker with full privileges is quite straightforward. You just have to pull the default image:

```
docker pull falcosecurity/falco:latest
```

Then run Falco with the following command:

```
docker run --rm -i -t \
    --privileged \
    -v /var/run/docker.sock:/host/var/run/docker.sock \
    -v /dev:/host/dev \
    -v /proc:/host/proc:ro \
    -v /boot:/host/boot:ro \
    -v /lib/modules:/host/lib/modules:ro \
    -v /usr:/host/usr:ro \
    -v /etc:/host/etc:ro \
    falcosecurity/falco:latest
```

The above command will install the driver on the fly before running Falco. The container image uses the Kernel Module by default. If you want to use the eBPF probe instead, just add the `-e FALCO_BPF_PROBE=""` option and remove `-v /dev:/host/dev` (only the Kernel Module requires */dev*).

As you can see, aside from the `--privileged` option, the above command mounts a set of paths from the host into the container (each `-v` option is a bind mount).

Specifically, the `-v /var/run/docker.sock:/host/var/run/docker.sock` option shares the Docker socket, so Falco can use Docker to obtain container metadata. (You may recall that Chapter 5 discusses Falco's data enrichment techniques). Add similar options for each container runtime available on your system. For example, if you also have Containerd, include `-v /run/containerd/containerd.sock:/host/run/containerd/containerd.sock`.

Falco requires sharing */dev* and */proc* to interface with the driver and the system, respectively. Other shared paths are needed to install the driver.

Least privileged mode

This running mode follows the principle of least privilege for enhanced security. Although this mode is the recommended way to run Falco in a container, it might not necessarily work in all systems and configurations. We advise you to give it a try anyway and fall back to the fully privileged mode only if this does not fit your environment.

As noted, this approach uses two different container images. The first step is to install the driver using the `falcosecurity/falco-driver-loader` image, a one-time step which requires full privileges. You need to install the driver before running Falco for the first time, and only when upgrading the driver. (Alternatively, as explained earlier, you can install the driver directly on the host using the `falco-driver-loader` script shipped with the binary package. If you did so, skip this step.)

To install the driver using a container image, pull the image first:

```
docker pull falcosecurity/falco-driver-loader:latest
```

Then run the installation command:

```
docker run --rm -i -t \
    --privileged \
    -v /root/.falco:/root/.falco \
    -v /proc:/host/proc:ro \
    -v /boot:/host/boot:ro \
    -v /lib/modules:/host/lib/modules:ro \
    -v /usr:/host/usr:ro \
    -v /etc:/host/etc:ro \
    falcosecurity/falco-driver-loader:latest
```

The above command installs the Kernel Module by default. If you want to use the eBPF probe instead, just add the `-e FALCO_BPF_PROBE=""` option.

The last step is to run Falco. Since the driver is already installed, you will just need to use the `falcosecurity/falco-no-driver` image. So, pull it first:

```
docker pull falcosecurity/falco-no-driver:latest
```

Then run Falco:

```
docker run --rm -i -t \
    -e HOST_ROOT=/ \
    --cap-add SYS_PTRACE --pid=host $(ls /dev/falco* | xargs -I
{} echo --device {}) \
    -v /var/run/docker.sock:/var/run/docker.sock \
    falcosecurity/falco-no-driver:latest
```

If you use another container runtime, customize the above command by adding a `-v` option accordingly.

Finally, there are some caveats when using the eBPF probe. You cannot use least privileged mode unless you have at least Kernel 5.8. This is because, with previous kernel versions, loading the eBPF probe required the `--privileged` flag. If you are running a kernel version equal to or greater than 5.8, you can use the SYS_BPF capability to overcome this issue by customizing the command as follows:

```
docker run --rm -i -t \
    -e FALCO_BPF_PROBE=""
    -e HOST_ROOT=/ \
    --cap-add SYS_PTRACE --cap-add SYS_BPF -pid=host \
    -v /root/.falco:/root/.falco \
    -v /var/run/docker.sock:/var/run/docker.sock \
    falcosecurity/falco-no-driver:latest
```

Note that on systems with the AppArmor LSM enabled, you will also need to pass the following: `--security-opt apparmor:unconfined`.

TIP

Depending on the Falco version you are using and your environment, you might need to customize the commands described in this section; refer to the online documentation.

Plugin Scenario

When using a plugin as your data source, there's no need to install a driver, nor will Falco need full privileges to run. However, since the container image contains the default Falco configuration file, you have to give Falco the required configuration for the plugin. You can do that by using an external configuration file and mounting it in the container.

So, as a preparation step, you have to create a local copy of the falco.yaml and modify it according to your plugin configuration. We will explain how to do that in the next chapter.

Once you have prepared your custom *falco.yaml*, to run Falco, use the following command:

```
docker run --rm -i -t \
    -v falco.yaml:/etc/falco/falco.yaml \
    falcosecurity/falco-no-driver:latest
```

If you want to use a plugin not shipped in the default Falco distribution, you will have to mount the plugin file in the container, too. For example, to mount a plugin called *libmyplugin.o*, add the following option to the above command:

```
-v libmyplugin.o:/usr/share/falco/plugins/libmyplugin.o
```

Deploying to a Kubernetes Cluster

One of the most common Falco use cases is securing clusters, so deploying Falco to Kubernetes is perhaps the most important installation method. The Falco Project recommends two main methods for that purpose:

Helm

> One quick and straightforward installation method uses Helm, a very popular tool to install and manage software built for Kubernetes. The Falco community provides and maintains a Helm Chart for Falco and other tools that integrate with Falco. Installing Falco using the provided chart is straightforward and mostly automatic.

Kubernetes manifest files

> The other installation method, mainly intended for giving high flexibility, is based on a set of Kubernetes manifest files. Those files provide default installation settings, and the users need to customize them based on their needs. Although this latter approach requires a bit more effort, it permits the installation of Falco virtually on any Kubernetes installation without the need for extra tools.

Both approaches are solid, and you should select one that best suits your environment and your organization’s requirements. In the following subsections, we will walk you through both. The only requirement is having a Kubernetes cluster installed and running.

NOTE

All of the installation methods for Kubernetes in this section use the Falco container image we discussed in the previous section, “Running in a container.”

Using Helm

If you prefer a fully automated installation process or are already using Helm in your environment, this installation method is for you. Having Helm installed is a prerequisite. Please follow their online documentation if you don’t have it installed yet.

Falco’s Helm chart will add Falco to all nodes in your cluster using a DaemonSet. Then each deployed Falco pod will try to install the driver on its own node. That’s the default configuration that reflects the most common scenario, syscall instrumentation.

TIP

Falco pods internally use `falco-driver-loader`, which tries to download a prebuilt driver; failing that, it will build the driver on the fly. Usually, no action is required. If you notice that the Falco pods are continuously restarting after being deployed, the process was probably unable to install the driver. This issue usually happens when a prebuilt driver is unavailable for your distribution or kernel and no kernel headers are available on the host. To build the driver, Kernel headers must be installed on the host. You can fix the issue by manually installing the kernel headers and then deploying Falco again.

Helm uses the Kubernetes context provided by kubectl to access your cluster. Before installing Falco with Helm, ensure that your local configuration points to the proper context. You can check that by running:

```
kubectl config current-context
```

If the context is not pointing to your targeted cluster or kubectl cannot access your cluster, you will have to address it. Otherwise, you can proceed with the next step.

Before installing the chart, add Falco's Helm repository so that your local Helm installation can find the Falco chart:

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

The preceding command is usually a one-time operation. To get the latest information about the Falco chart, use:

```
helm repo update
```

Use the preceding command whenever you want to install and update Falco with Helm.

The next and final step is actually to install the chart by running:

```
helm install falco falcosecurity/falco
```

The chart installs the Kernel Module by default. Optionally, if you want to use the eBPF probe instead, just append `--set ebpf.enabled=true` to the above command.

And you're done. After a while, Falco's pods will show up in your cluster. You can use the following command to check whether they are ready:

```
kubectl get all
```

The chart installs Falco for the default scenario (syscalls instrumentation), as per the default settings. The Helm installation process for other scenarios is very similar; just provide a different configuration. We will discuss how to customize your Falco deployment in Chapter 9. You can find more information about Falco's chart configuration in its online documentation.

Using Manifests

Kubernetes manifests are JSON or YAML files (mainly YAML) that contain the specifications for one or more Kubernetes API objects and describe your application and its configurations. The kubectl command-line utility lets you deploy your workload in Kubernetes using these files. Projects often provide almost-ready-to-use example manifests, but you'll usually need to adapt them to your needs.

Since Falco supports very different scenarios and environments, the Falco Project does not officially provide manifests for all use cases. However, for the syscall instrumentation scenario, you can use its example manifests[1] (listed in Table 8-2) as a starting point to make your customized manifests. In this section, we will give you an overview of the structure of Falco manifests, so that you can work with them. Then we will show you how to use them with kubectl to deploy Falco in a Kubernetes cluster.

Table 8-2. Example manifest files f

o
r

F
a
l
c
o

Filename	Description
daemonset.yaml	Specify a DaemonSet so that a copy of the Falco pod will run on each node. That's required by the syscall instrumentation scenario. The Pod specification uses the falcosecurity/falco container images. It also includes all settings to run the image in this scenario, similar to those described in the "Running in a container" section.
configmap.yaml	Specify a ConfigMap containing the default *falco.yaml* file and rule set files. Modify it according to your needs.
serviceaccount.yaml	Specify a ServiceAccount for running Falco's pods. Falco requires it to talk with the Kubernetes API. You usually don't need to change it, unless you want to change the service account name.
clusterrole.yaml	Specify a ClusterRole, including the RBAC authorizations required by Falco to talk with the Kubernetes API. Don't change the list of permissions needed, or Falco will not enrich the Kubernetes metadata correctly.
clusterrolebinding.yaml	Specify a ClusterRoleBinding that grants the permissions defined in *clusterole.yaml* to the service account defined in *serviceaccont.yaml*. You usually won't need to change this, unless you've changed the service account or the cluster role name in the other files.

Once you have modified the manifest files according to your needs, to apply them to Kubernetes (that is, to deploy Falco to Kubernetes), just run the following command:

```
kubectl apply \
    -f ./templates/serviceaccount.yaml \
    -f ./templates/clusterrole.yaml \
    -f ./templates/clusterrolebinding.yaml \
    -f ./templates/configmap.yaml \
    -f ./templates/daemonset.yaml
```

Falco's pods should show up in your cluster after a while. To check whether they are ready, use:

```
kubectl get all
```

If everything went well, Falco is now up and running in your production cluster—and you have learned how to customize your Falco deployment. Congratulations!

[1] The actual URLs of Falco manifest example files for Kubernetes may change from time to time, but you can always find a link to them in the official documentation. Falco's Helm chart can generate those files, too. Surprisingly, the Falco Project uses this Helm functionality to automatically publish up-to-date manifest example files under Falcosecurity's GitHub organization.

Chapter 9. Prospective Table of Contents (Subject to Change)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at sgrey@oreilly.com.

12. Collecting Falco Events
13. Falco Rules
14. Falco Development
15. How to Contribute

About the Authors

Loris Degioanni is the CTO and founder of Sysdig. He is also the creator of the popular open source troubleshooting tool sysdig, and the CNCF runtime security tool Falco. Prior to founding Sysdig, Loris was one of the original contributors to Wireshark, the open source network analyzer. Loris holds a PhD in computer engineering from Politecnico di Torino and lives in Davis, California.

Leonardo Grasso is a Falco core maintainer and an open source software engineer at Sysdig. He primarily takes care of Falco and spends the rest of his time contributing to various projects. Leonardo has a strong passion for software design and has long professional experience in the R&D field. He currently lives in Milan, Italy.