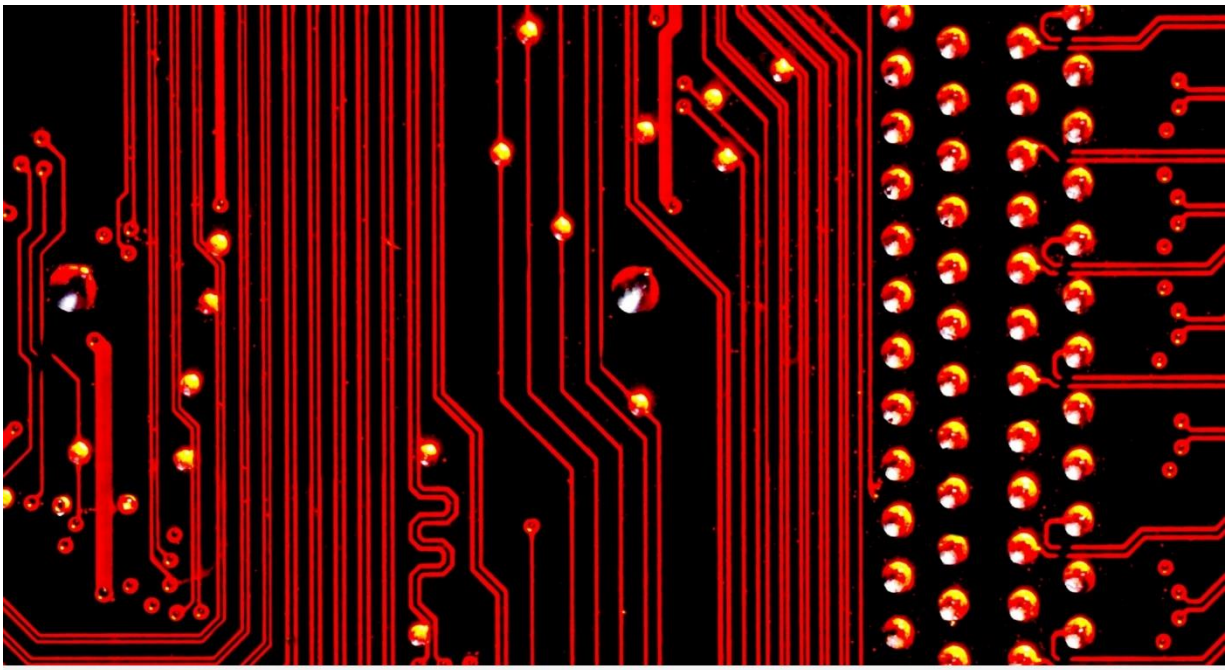
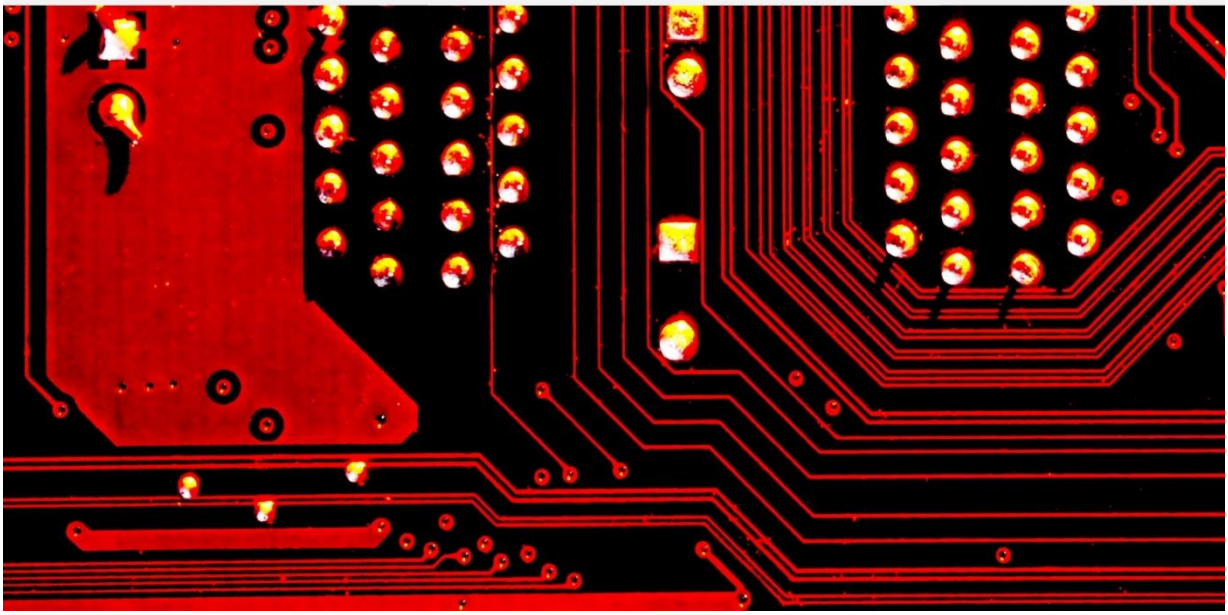


DETECTING THREATS IN REALTIME WITH FALCO

A white paper by Pranab Tandon



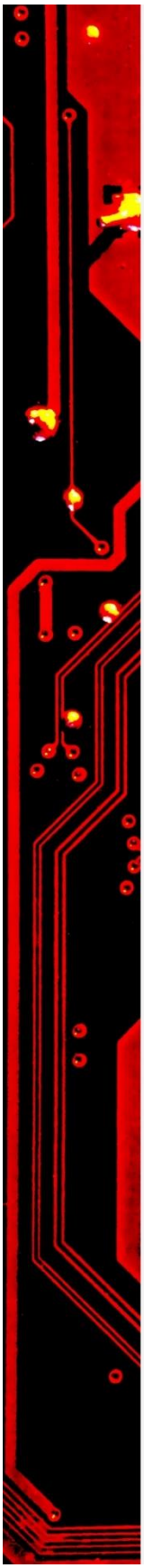
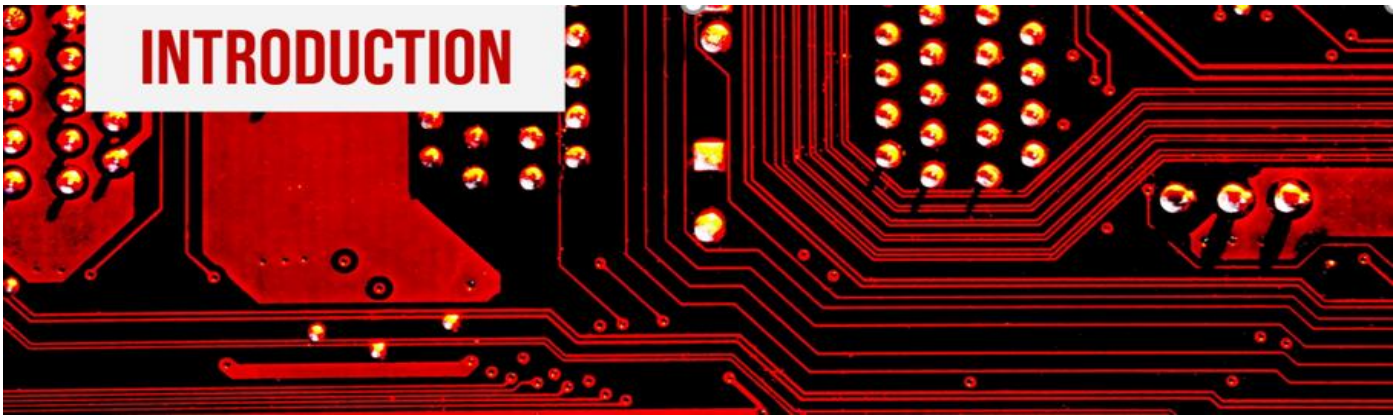


TABLE OF CONTENTS



Chapter 1 Introduction to Falco

What is Falco ?

Falco is a versatile tool for threat and risk detection in containers, Kubernetes clusters, on a Linux host and in public/private clouds. It can detect threats to system and alerts the user in real-time when it finds one. In very simple terms it can be seen as a generic virus/malware/APT detection tool, most modern viruses/malware or advance persistent threats try to do a set of things to sabotage or own a system , most of these are achieved using system calls in one way or the other, Falco detects these system calls and alerts the administrator of attack in progress so that corrective actions can be taken, Falco can easily detect threats including but not limited to:

- Privilege escalations
- A shell is running inside a container or pod in Kubernetes.
- Access to sensitive data
- A standard system binary, such as ls, is making an outbound network connection.
- Ownership and Mode changes
- A privileged pod is started in a Kubernetes cluster.
- Unexpected network connections or socket mutations
- Unwanted program execution
- A non-device file is written to /dev.
- Data exfiltration
- Compliance violations
- A container is running in privileged mode, or is mounting a sensitive path, such as /proc, from the host.
- A server process is spawning a child process of an unexpected type.
- Unexpected read of a sensitive file, such as /etc/shadow.
- Namespace changes using tools like setns
- Read/Writes to well-known directories such as /etc, /usr/bin, /usr/sbin, etc

- Creating symlinks
- Spawned processes using execve
- Executing SSH binaries such as ssh, scp, sftp, etc
- Mutating Linux coreutils executables
- Mutating login binaries
- Mutating shadowutil or passwd executables such as shadowconfig, pwck, chpasswd, getpasswd, change, useradd, etc, and others

Falco excels at detecting threats, intrusions, and data breaches at runtime and in real time. Out of box supports is available for containers, Kubernetes, and cloud infrastructures. It can secure both workloads(containers, processes, services) and infrastructure (hosts, VMs, network, cloud infrastructure and services).The beauty of the architecture of the Falco is it is lightweight, efficient, and scalable without impacting the performance. It can detect many classes of threats, additionally you can customize it based on your current threat scenario.

Chapter 2 Falco a 1000-foot view

How Falco Works High Level Overview

Picture a typical security surveillance system in a building, what does come to your mind.

1. Video Camera
2. Motion detection
3. Fire detection
4. Forced entry detection
5. Biometric scanners

The list of these goes on and on. All these are kind of sensors which sense certain condition and inform or alert when any threat is detected. These sensors like video camera, fire detection sense inputs from certain sources and then translate them into useful actionable information and operator can be alerted. Falco works on similar principles. Here is a simple architecture of Falco, we will dive deep later.

Sensors : A sensor is nothing but a rule matching 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 generated. In a typical deployment scenario, there are multiple sensors which are spread across all the nodes which are being monitored which generates threat alerts in Realtime.

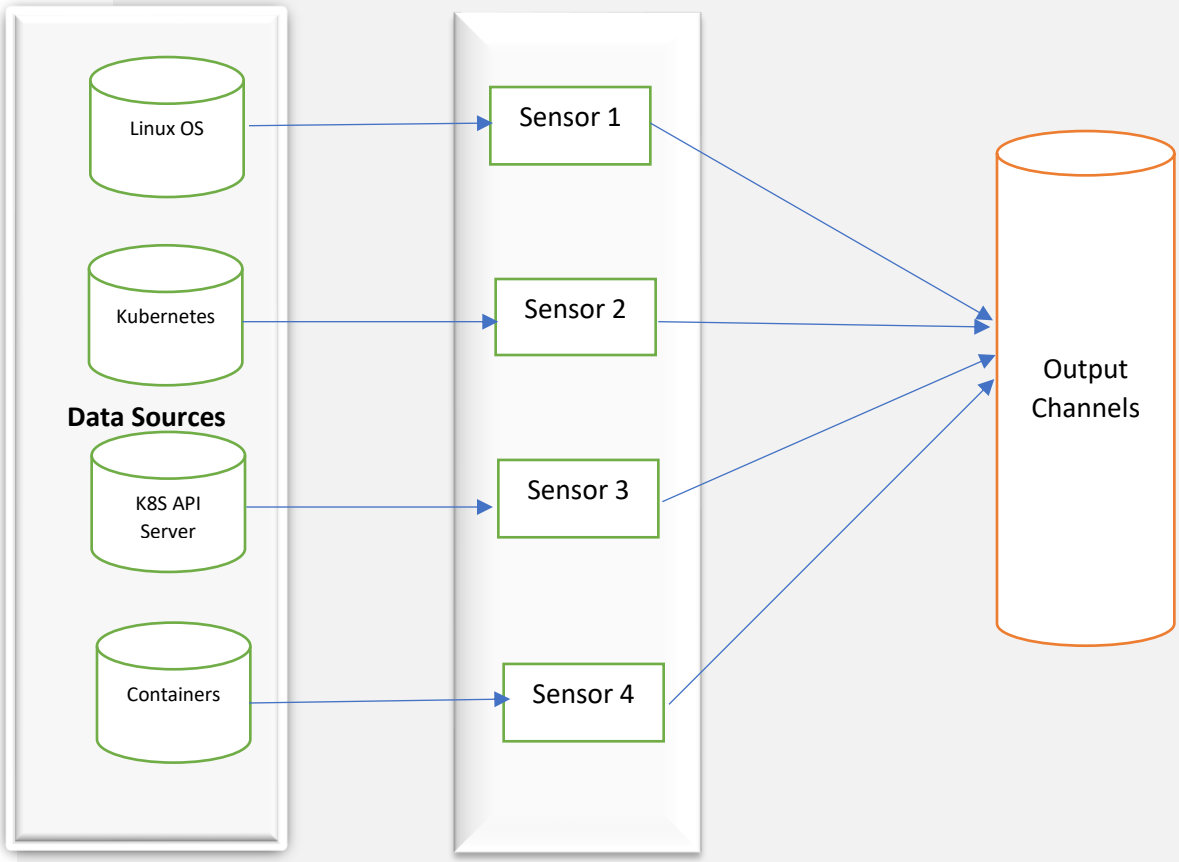


Figure 1: High Level Architecture of Falco

Data Sources: Each sensor can collect input data from various sources. Initially Falco was designed to exclusively operate on system calls but now there are other input sources like container, Kubernetes audit logs, CloudTrail etc. Falco's data sources can be grouped into two main categories:

1. System calls
2. Plugins

System calls are very important data source for us. System calls are request to Operating system kernel which are made by running program to achieve its functionality like 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. *In a telco environment it is best to use kernel module as most of the Telecom applications are I/O extensive and eBPF module may not provide the kind of performance required in these environments.*

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. Plugins are relatively new and were added in 2021. It connects a varied number of inputs to Falco, such as cloud logs. Plugins are shared libraries that conform to a documented API and allow for:

a) adding new event sources that can be evaluated using filtering expressions/Falco rules.
b) adding the ability to define new fields that can extract information from events.

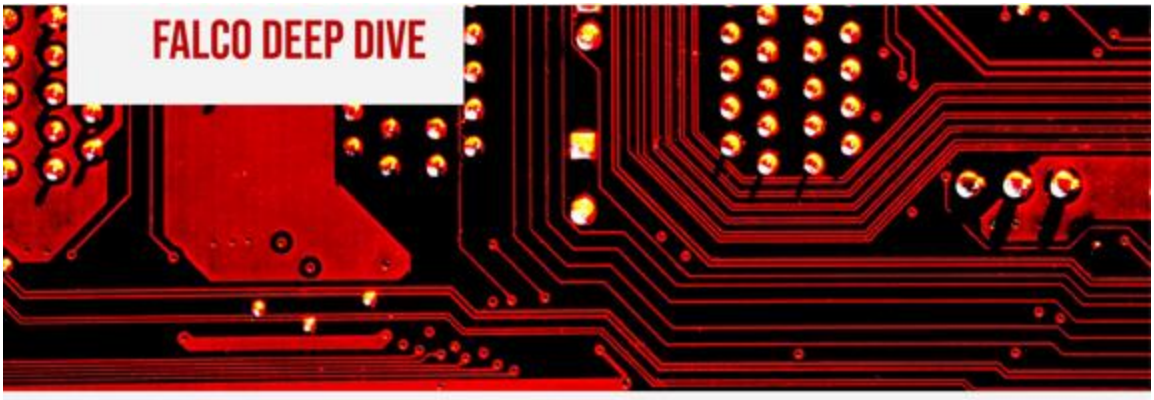
Currently, the available plugins are:

(1) CloudTrail: Reads CloudTrail JSON logs from files/S3 and injects as events.
(2) dummy: Reference plugin use to document plugins interface.
(3) dummy: Like dummy but written in C++.
(4) json: Extracts values from any JSON payload.

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. It is possible to enable more than one options simultaneously. It is easy to spot them within the configuration file 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. Falco comes with six built-in output channels; they are listed in the table below.

Channel Name	Description
Standard Output	Send notifications to the Falco's standard output (i.e., stdout)
A File	When configured to send alerts to a file, a message is written to the file for each alert.

Syslog	When configured to send alerts to syslog, a syslog message is sent for each alert. The actual format depends on your syslog daemon.
A spawned program	When configured to send alerts to a program, Falco starts the program for each alert and writes its contents to the program's standard input.
A HTTP[s] end point	Post notification to an URL. If you'd like to send alerts to an HTTP[s] endpoint, you can use the `http_output`
A client via the gRPC API	Sends alerts to an external program connected via gRPC API.



Chapter 2 Falco a deep dive

Falco's Architecture

Falco is composed of three main components and one upcoming component:

1. Userspace program - is the CLI tool falco that you can use to interact with Falco. The userspace program handles signals, parses information from a Falco driver, and sends alerts.
2. Configuration - defines how Falco is run, what rules to assert, and how to perform alerts.
3. Driver - is a software that adheres to the Falco driver specification and sends a stream of system call information. You cannot run Falco without installing a driver. Currently, Falco supports the following drivers:
    a) (Default) Kernel module built on libscap and libsinsp C++ libraries
    b) BPF probe built from the same modules
    c) Userspace instrumentation

4. Plugins - allow users to extend the functionality of falco libraries/falco executable by adding new event sources and new fields that can extract information from events.

The figure below shows the architecture of the Falco tool.

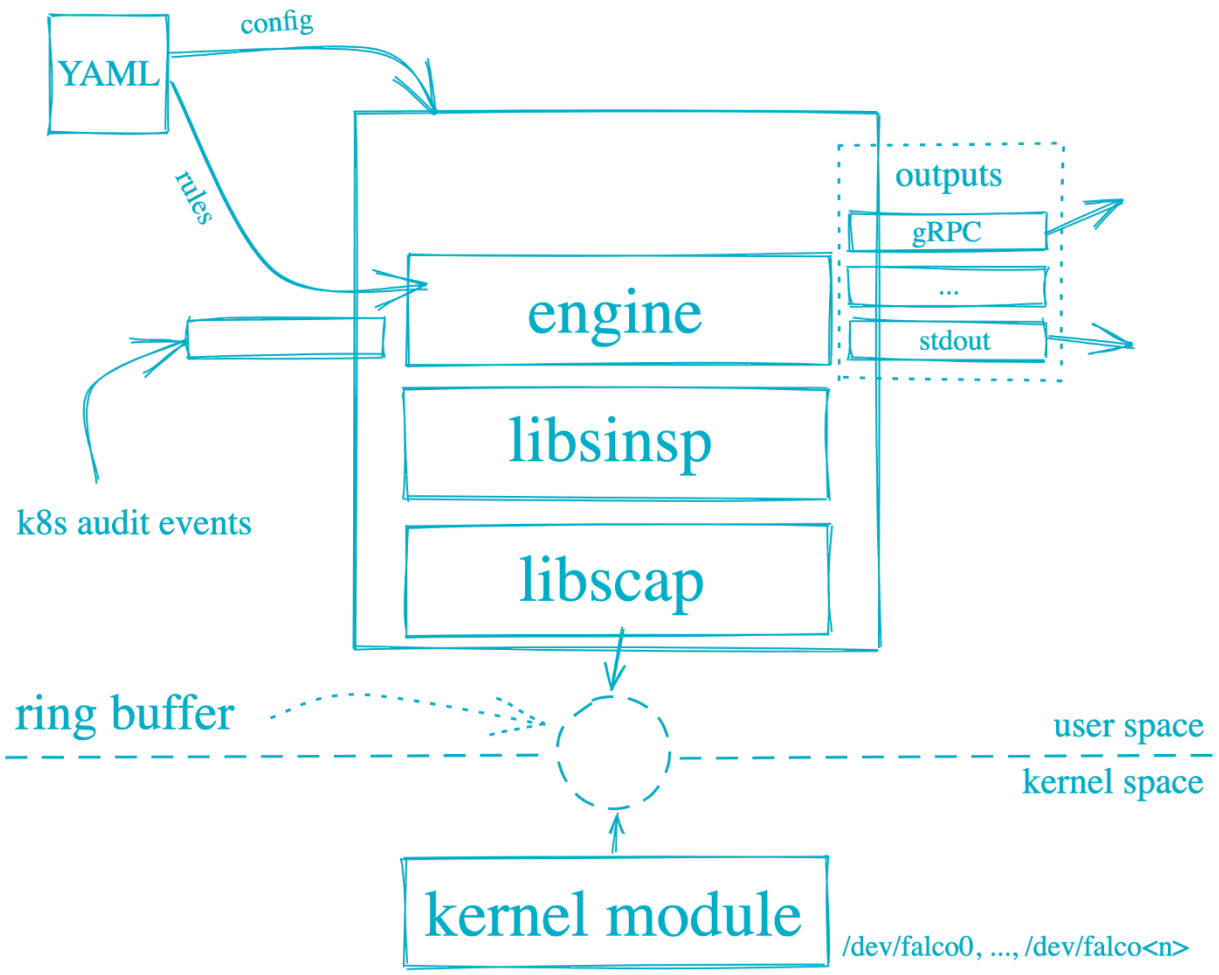


Figure 2 : Falco Architecture

Falco as a tool spreads across both the userspace and kernel space. Let us understand the various libraries, components in the figure above and how to they work together to provide excellent cloud native security. Falco works by taking Linux system call information at runtime and rebuilding the state of the kernel in memory. The Falco engine depends on a driver to consume the raw stream of system call information. The Falco Kernel module is the traditional way of consuming the required stream of data from the kernel.

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. Currently the Falco project supports 3 different drivers in which the engine can consume this information.

- A kernel module
- An eBPF probe

The kernel module works with any version of the Linux kernel, including older ones. It introduces very little overhead in tems of compute resources , 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.

Ring buffer

System call data captured from Kernel space has to be moved to userspace for data enrichment and further processing, ring buffer support zero copy architecture which efficiently transfers this data from the kernel to user space, where libscap will receive it.

Libscap

Libscap stands for "library for system capture,". libscap is the gate through which the input data goes before getting into the Falco processing pipeline. The libscap controls 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.

libsinsp

libsinsp stands for "library for system inspection." This is where each event is parsed, inspected, and is evaluated against a set of user-specified filters. Libsinsp taps into the stream of data libscap produces, enriches it, and offers several higher-level primitives to work with it.

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(/etc/falco/falco_rules.yaml)
- 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.

Important YAML Files

When Falco is started, it loads several files. It first loads the configuration file.

```
Wed May 5 14:16:03 2022: 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 required, 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 you can customize predefined rules by simply overriding them in files that appear later in the list.

Falco Examples

Here are some examples of the types of behavior falco can detect.

For a more comprehensive set of examples, see the full rules file at falco_rules.yaml

a) A shell is run in a container

```
- macro: container
  condition: container.id != host

- macro: spawned_process
  condition: evt.type = execve and evt.dir=<

- rule: run_shell_in_container
  desc: a shell was spawned by a non-shell program in a container. Container
entrypoints are excluded.
  condition: container and proc.name = bash and spawned_process and
proc.pname exists and not proc.pname in (bash, docker)
  output: "Shell spawned in a container other than entrypoint
(user=%user.name container_id=%container.id container_name=%container.name
shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline)"
  priority: WARNING
```

b) Unexpected outbound Elasticsearch connection

```
- macro: outbound
  condition: syscall.type=connect and evt.dir=< and (fd.typechar=4 or
fd.typechar=6)

- macro: elasticsearch_cluster_port
  condition: fd.sport=9300

- rule: elasticsearch_unexpected_network_outbound
  desc: outbound network traffic from elasticsearch on a port other than the
standard ports
  condition: user.name = elasticsearch and outbound and not
elasticsearch_cluster_port
  output: "Outbound network traffic from Elasticsearch on unexpected port
(connection=%fd.name)"
  priority: WARNING
```

Configuration files

TLDR;

Configuration for the Falco daemon is done by /etc/falco/falco.yaml , it supports following functionality

Config	Type	Description
rules_file	List	The location of the rules file(s). This can contain one or more paths to separate rules files. The following examples are equivalent: rules_file: - path1 - path2 rules_file: [path1, path2]
time_format_iso_8601	Boolean	If true (default is false), the times displayed in log messages and output messages will be in ISO 8601. By default, times are displayed in the local time zone, as governed by /etc/localtime.
json_output	Boolean	Whether to use JSON output for alert messages.
json_include_output_property	Boolean	When using json output, whether or not to include the output property itself (e.g. File below a known binary directory opened for writing (user=root ....) in the JSON output.

log_stderr	Boolean	If true, log messages describing Falco's activity will be logged to stderr. Note these are *not* alert messages---these are log messages for Falco itself.
log_syslog	Boolean	If true, log messages describing Falco's activity will be logged to syslog.
log_level	Enum with the following possible values: emergency, alert, critical, error, warning, notice, info, debug	Minimum log level to include in logs. Note: these levels are separate from the priority field of rules. This refers only to the log level of Falco's internal logging.
priority	Enum with the following possible values: emergency, alert, critical, error, warning, notice, info, debug	Minimum rule priority level to load and run. All rules having a priority more severe than this level will be loaded/run.

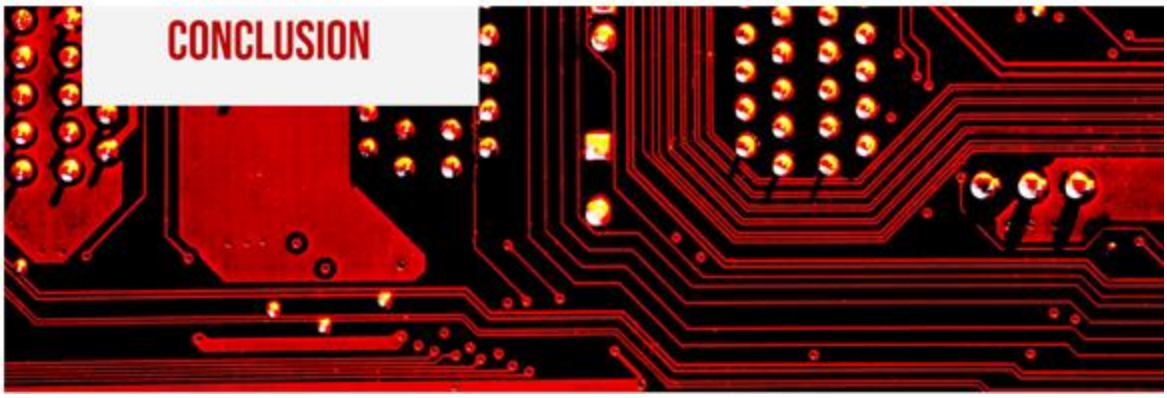
syscall_event_drops	List containing the following sub-keys:
	• actions: A list containing one or more of these boolean sub-keys:
	  ○ ignore: do nothing. If an empty list is provided, ignore is assumed.
	  ○ log: log a CRITICAL message noting that the buffer was full.
	  ○ alert: emit a Falco alert noting that the buffer was full.
	  ○ exit: exit Falco with a non-zero rc.
	• rate: The steady-state rate at which actions can be taken. Units of actions/second. Default 0.03333 (one action per 30 seconds).
	• max_burst: The maximum number of actions that can be taken before the steady-state rate is applied.
		Controls Actions For Dropped System Call Events.

buffered_outputs	Boolean	Whether or not output to any of the output channels below is buffered. Defaults to false.
outputs	List containing the following sub-keys: • rate: <notifications/second> • outputs: max_burst: <number of messages>	A throttling mechanism implemented as a token bucket limits the rate of Falco notifications. This throttling is controlled by the rate and max_burst options. rate is the number of tokens (i.e. right to send a notification) gained per second, and defaults to 1. max_burst is the maximum number of tokens outstanding, and defaults to 1000. With these defaults, Falco could send up to 1000 notifications after an initial quiet period,

		and then up to 1 notification per second afterward. It would gain the full burst back after 1000 seconds of no activity.
syslog_output	List containing the following sub-keys: • enabled: [true|false]	If true, Falco alerts will be sent via syslog.
file_output	List containing the following sub-keys: • enabled: [true|false] • keep_alive: [true|false] • filename: <path>	If enabled is set to true, Falco alerts will be sent to the filepath specified in filename. If keep_alive is set to false (the default), Falco will re-open the file for every alert. If true, Falco will open the file once and keep it open for all alerts. It may also be necessary to specify --unbuffered using the Falco CLI.
stdout_output	List containing the following sub-keys: • enabled: [true|false]	If enabled is set to true, Falco alerts will be sent to standard output (stdout).
program_output	List containing the following sub-keys: • enabled: [true|false] • keep_alive: [true|false]	If enabled is set to true, Falco alerts will be sent to a program. If keep_alive is set to false (the default), run the program for each alert. If true, Falco will spawn the program once and keep it open for all alerts. It may also be necessary to specify --unbuffered using the Falco CLI. The program setting specifies the program to be run for each alert. This is started via the shell, so you can specify a command pipeline to allow for additional formatting.
http_output	List containing the following sub-keys: • enabled: [true|false] • url: [http[s]://path/to/webhook/]	As of 0.15.0, if enabled is set to true, Falco alerts will be sent to the HTTP[s] URL defined by url. Currently this is a blocking operation, and this output does not support keep_alive.
webserver	List containing the following sub-keys: • enabled: [true|false] • listen_port • k8s_audit_endpoint • ssl_enabled: [true|false] • ssl_certificate: <path>	If enabled is set to true, Falco will start an embedded web server to accept Kubernetes audit events. listen_port specifies the port on which the web server will listen. The default is 8765. k8s_audit_endpoint specifies the URI on which to listen for Kubernetes audit events. The default is /k8s-audit. ssl_enabled enables SSL support for the webserver, using the certificate specified in ssl_certificate. The specified ssl_certificate file

		should contain the server's certificate as well as the key as documented by civitweb.
grpc	List containing the following sub-keys: • enabled: [true|false] • bind_address: [unix://<path>.sock|address:port] • threadiness: <integer> • private_key: <path> • cert_chain: <path> • root_certs: <path>	If enabled is set to true, Falco will embed a gRPC server to expose its gRPC API. The default is false. Falco supports running a gRPC server with two main binding types: • Over a local unix socket with no authentication • Over the network with mandatory mutual TLS authentication (mTLS) bind_address specifies either the unix socket path or the address and port on which the gRPC server will listen. The default is unix:///var/run/falco.sock. threadiness defines the number of threads to use to serve gRPC requests. When threadiness is 0, Falco automatically guesses it depending on the number of online cores. The default is 0. The gRPC server over the network can only be used with mutual authentication between the clients and the server using TLS certificates, and the following options should be provided: private_key path of the private key for server authentication. The default is /etc/falco/certs/server.key. cert_chain path of the public cert for server authentication. The default is /etc/falco/certs/server.crt. root_certs path of the CA certificate (or chain) common between the server and all the clients. The default is /etc/falco/certs/ca.crt. How to generate the certificates is documented here. Please always remember that the only common thing between server and clients is the root certificate. Every client will need to generate their own certificates signed by the same root CA as the server.
grpc_output	List containing the following sub-keys: • enabled: [true|false]	If enabled is set to true, Falco will start collecting outputs for the gRPC server. It's important to consume them with an output client. Example of output client here.

plugins	A list of objects. Each object has the following sub-keys: • name: <string> • library_path: <path> • init_config: [<string>|<object>] • open_params: <string>	Defines the set of plugins Falco can load. name provides the plugin name. This is only used for load_plugins, but by convention should be the same as the value returned by the plugin_get_name() api function. library_path is path to the shared library. The path can be relative, in which case it is relative to Falco's "share" directory under a "plugins" subdirectory e.g. /usr/share/falco/plugins. If init_config is present, it contains the configuration that will be provided as an argument to the init() function. This can either be in string or object form. If a string is used, the plugin is configured with the string content as-is. If an object is used, by convention its contents are formatted into a JSON string and then passed to the plugin. In both cases, the configuration string is automatically parsed against a schema before being passed to the plugin if the get_init_schema function is implemented, otherwise it gets directly passed as an opaque string. if open_params: is present, it contains the exact params string that will be provided as an argument to the open() function, which is used in source plugins only.
load_plugins	A list of plugin names, corresponding to the name sub-key from the plugins config item. Example: load_plugins: [cloudtrail, json]	Defines the set of plugins that will actually be loaded. This is optional--if the property is not present, all plugins named in the plugins list will be loaded.



Conclusion

In the world of security there are no Swiss army knives. Most of the tools available can support in solving a part of the security problem. It is important to know what the tool you are using can do and what it cannot. Falco is not a general-purpose policy language: it doesn't offer the articulateness of a regular programming language and cannot perform correlation across different engines. Its rule engine, instead, is designed to apply stateless rules at a very fast pace in many places around your infrastructure. If you are looking for a powerful centralized policy language, we suggest you look at The Open Policy Agent (OPA, pronounced "oh-pa")which is an open source, general-purpose policy engine that unifies policy enforcement across the stack. OPA provides a high-level declarative language that lets you specify policy as code and simple APIs to offload policy decision-making from your software. You can use OPA to enforce policies in microservices, Kubernetes, CI/CD pipelines, API gateways, and more.

Falco is not designed to perform analytics on a centrally stored data. Rule validation is performed at the endpoint and only the alerts are sent to a centralized location. Falco does not offer deep packet inspection for network payloads. Therefore, it is not suited for implementing layer 7 security policies. Off the shelf network-based intrusion detection system (IDS)/L7 firewall are better suited for such a use case.

Reference

- https://falco.org/docs/
- https://www.openpolicyagent.org/docs/latest/
- Practical Cloud Native Security with Falco by Loris Degioanni and Leonardo Grasso

Contact me

If you have any question comments or queries , please feel free to get in touch with me on pranabtandon[at]gmail.com

LinkedIn: in.linkedin.com/pub/pranab-tandon/a/598/784/