SysFlow Telemetry Pipeline

Release 0.4

The SysFlow team

Jan 15, 2024

CONTENTS:

The SysFlow Telemetry Pipeline is a framework for monitoring cloud and enterprise workloads. The framework builds the plumbing required for system telemetry so that users can focus on writing and sharing analytics on a scalable, common open-source platform.

Note: If in a hurry, skip to our quick start guide.

The backbone of the telemetry pipeline is a new data format which lifts raw system event information into an abstraction that describes process behaviors, and their relationships with containers, files, and network activity. This object-relational format is highly compact, yet it provides broad visibility into legacy endpoints and container clouds.

The platform is designed as a pluggable edge processing architecture which includes a policy engine that accepts declarative policies that support edge filtering, tagging, and alerting on SysFlow streams. It also offers several APIs that allow users to process SysFlow with their favorite toolkits.

The pipeline can be deployed using Docker, Kubernetes, OpenShift, and bare metal/VMs. The SysFlow agent can be configured as an edge analytics pipeline to stream SysFlow records through rsyslog, or as a batch exporter of raw SysFlow traces to S3-compatible object stores.

An integrated Jupyter environment makes it easy to perform log hunting on collected traces. There are also Apache Avro schema files for SysFlow so that users can generate APIs for other programming languages. C++, Python, and Golang APIs are available, allowing users to interact with SysFlow traces programmatically.

To learn more about SysFlow, check the table of contents below.

We welcome feedback, bug reports, and contributions!

CHAPTER

ONE

KEEP IN TOUCH

Please connect with us on our Slack community!

CHAPTER

TWO

BUGS & FEATURE REQUESTS

For bugs and feature requests, please check our issue tracker.

CHAPTER

THREE

LICENSE

SysFlow and all projects are released under the Apache v2.0 license.

3.1 Quick Start

We encourage you to check the documentation first, but here are a few tips for a quick start.

3.1.1 Deployment options

The SysFlow agent can be deployed in batch or edge processing export configurations. In the batch configuration, SysFlow exports the collected telemetry as trace files (batches of SysFlow records) to any S3-compliant object storage service.

In edge processing configuration, SysFlow exports the collected telemetry as events streamed to a rsyslog collector or Elasticsearch. This deployment enables the creation of customized edge pipelines, and offers a built-in policy engine to filter, enrich, and alert on SysFlow records.

Instructions for `Docker Compose`, `Helm`, and `binary package` deployments of complete SysFlow stacks are available here.

3.1.2 Inspecting collected traces

A command line utilitiy is provided for inspecting collected traces or convert traces from SysFlow's compact binary format into human-readable JSON or CSV formats.

```
docker run --rm -v /mnt/data:/mnt/data sysflowtelemetry/sysprint /mnt/data/<trace>
```

where `trace` is the the name of the trace file inside `/mnt/data`. If empty, all files in `/mnt/data` are processed. By default, the traces are printed to the standard output with a default set of SysFlow attributes. For a complete list of options, run:

```
docker run --rm -v /mnt/data:/mnt/data sysflowtelemetry/sysprint  -h
```

This command line tool can also be installed directly on the host using pip.

```
python3 -m pip install sysflow-tools
```

3.1.3 Analyzing collected traces

A Jupyter environment is available for inspecting and implementing analytic notebooks on collected SysFlow data. It includes APIs for data manipulation using Pandas dataframes and a native query language (`sfql`) with macro support. To start it locally with example notebooks, run:

```
git clone https://github.com/sysflow-telemetry/sf-apis.git && cd sf-apis
docker run --rm -d --name sfnb -v $(pwd)/pynb:/home/jovyan/work -p 8888:8888␣
↪sysflowtelemetry/sfnb
```

Then, open a web browser and point it to `http://localhost:8888` (alternatively, the remote server name or IP where the notebook is hosted). To obtain the notebook authentication token, run `docker logs sfnb`.

3.2 SysFlow Specification

The SysFlow format lifts raw system event information into an abstraction that describes process behaviors, and their relationships with containers, files, and network. This object-relational format is highly compact, yet it provides broad visibility into container clouds. The framework includes several APIs that allow users to process SysFlow with their favorite toolkits.

3.2.1 Overview

Figure 1 shows a diagram of the SysFlow format.

Entities represent the components on a system that we are interested in monitoring. We currently support four types of entities: Pods, Containers, Processes, and Files. As shown in Figure 1, Containers contain references to both Pods, Processes and Files, and the four are linked through object identifiers (more on this later).

Entity behaviors are modeled as events or flows. Events represent important individual behaviors of an entity that are broken out on their own due to their importance, their rarity, or because maintaining operation order is important. An example of an event would be a process clone or exec, or the deletion or renaming of a file. By contrast, a Flow represents an aggregation of multiple events that naturally fit together to describe a particular behavior. For example, we can model the network interactions of a process and a remote host as a bidirectional flow that is composed of several events, including connect, read, write, and close. SysFlow also model events generated by the Kubernetes controller.

SysFlow enables users to configure the granularity of system-level data desired based on resource limitations and data analytics requirements. In this way, behaviors can be broken out into individual events or combined into smaller aggregated volumetric flows. The current specification describes events and flows in three key behavioral areas: Files, Networks, and Processes.

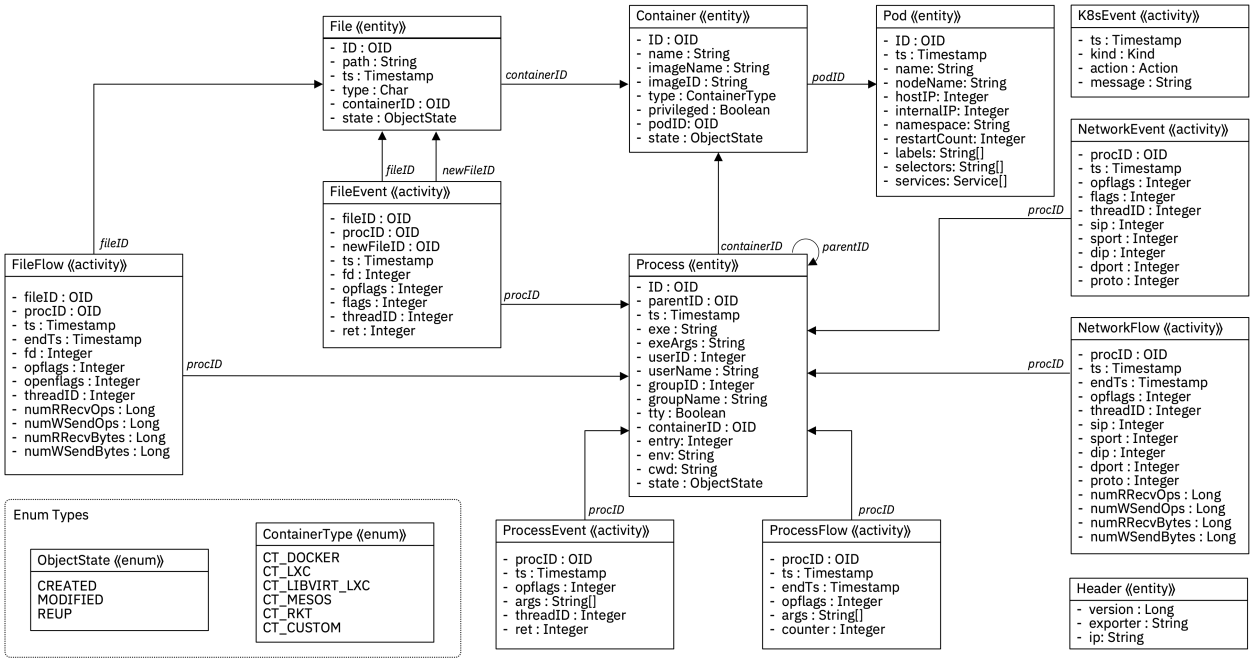

Figure 1: SysFlow Object Relational View

Entities

Entities are the components on a system that we are interested in monitoring. These include pods, containers, processes, and files. We also support a special entity object called a Header, which stores information about the SysFlow version, and a unique ID representing the host or virtual machine monitored by the SysFlow exporter. The header is always the first record appearing in a SysFlow File. All other entities contain a timestamp, an object ID and a state. The timestamp is used to indicate the time at which the entity was exported to the SysFlow file.

Object ID

Object IDs allow events and flows to reference entities without having duplicate information stored in each record. Object IDs are not required to be globally unique across space and time. In fact, the only requirement for uniqueness is that no two objects managed by a SysFlow exporter can have the same ID simultaneously. Entities are always written to the binary output file before any events, and flows associated with them are exported. Since entities are exported first, each event, and flow is matched with the entity (with the same id) that is closest to it in the file. Furthermore, every binary output file must be self-contained, meaning that all entities referenced by flows/events must be present in every SysFlow file generated.

State

The state is an enumeration that indicates why an entity was written to disk. The state can currently be one of three values:

State	Description
CREATED	Indicates that the entity was recently created on the host/VM. For example, a process clone.
MODIFIED	Indicates that some attributes of the entity were modified since the last time it was exported.
REUP	Indicates that the entity already existed, but is being exported again, so that output files can be self-contained.

Each entity is defined below with recommendations on what to use for object identifiers, based on what is used in the current implementation of the SysFlow exporter.

Header

The Header entity is an object which appears at the beginning of each binary SysFlow file. It contains the current version of SysFlow as supported in the file, and the exporter ID.

Attribute	Type	Description	Since (schema version)
version	long	The current SysFlow version.	1
exporter	string	Globally unique id representing the host monitored by SysFlow.	1
ip	string	IP address in dot notation representing the monitored host.	2

Container

The Container entity represents a system or application container such as docker or LXC. It contains important information about the container including its id, name, and whether it is privileged.

Attribute	Type	Description	Since (schema version)
id	string	Unique string representing the Container Object as provided by docker, LXC, etc.	1
state	enum	state of the process (CREATED, MODIFIED, REUP).	not implemented
timestamp (ts)	int64	The timestamp when container object is exported (nanoseconds).	not implemented
name	string	Container name as provided by docker, LXC, etc.	1
image	string	Image name associated with container as provided by docker, LXC, etc.	1
imageID	string	Image ID associated with container as provided by docker, LXC, etc.	1
type	enum	Can be one of: CT_DOCKER, CT_LXC, CT_LIBVIRT_LXC, CT_MESOS, CT_RKT, CT_CUSTOM	1
privileged	boolean	If true, the container is running with root privileges	1

Pods

The Pod entity represents a logical aggregation of containers in Kubernetes. It contains metadata about k8s pod including its id, name, and host and internal IPs.

Attribute	Type	Description	Since (schema version)
id	string	Unique string representing the Pod Object as provided by k8s or OpenShift	4
timestamp (ts)	int64	The timestamp when pod object is exported (nanoseconds)	4
name	string	Pod name	4
nodeName	string	None name	4
hostIP	int32	Host IP address (the exposed IP address of the Pod)	4
internalIP	int32	Internal Pod IP address	4
namespace	string	Namespace in which the pod runs	4
restartCount	int64	Number of restarts that have occurred for the pod	4
labels	string[]	Labels associated with the pod	4
selectors	Selector[]	K8s selectors associated with the pod	4
services	Service[]	K8s services associated with the pod	4

Process

The process entity represents a running process on the system. It contains important information about the process including its host pid, creation time, oid id, as well as references to its parent id. When a process entity is exported to a SysFlow file, all its parent processes should be exported before the process, as well as the process's Container entity. Processes are only exported to a SysFlow file if an event or flow associated with that process or any of its threads are exported. Threads are not explicitly exported in the process object but are represented in events and flows through a thread id field. Finally, a Process entity only needs to be exported to a file once, unless it's been modified by an event or flow.

> **NOTE** In current implementation, the creation timestamp is the time at which the process is cloned. If the process was cloned before capture was started, this value is 0. The current implementation also has problems getting absolute paths for exes when relative paths are used to launch processes.

Attribute	Type	Description	Since (schema version)
state	enum	state of the process (CREATED, MODIFIED, REUP)	1
OID: *host pidcreate ts*	**struct** *int64int64*	The Process OID contains the host pid of the project, and creation timestamp.	1
POID: *parent host pidparent create ts*	**struct** *int64int64*	The OID of the parent process can be NULL if not available or if a root process.	1
timestamp (ts)	int64	The timestamp when process object is exported (nanoseconds).	1
exe	string	Full path (if available) of the executable used in the process launch; otherwise, it's the name of the exe.	1
exeArgs	string	Concatenated list of args passed on process startup.	1
uid	int32	User ID under which the process is running.	1
userName	string	User name under which the process is running.	1
gid	int32	Group ID under which the process is running	1
groupName	string	Group Name under which the process is running	1
tty	boolean	If true, the process is tied to a shell	1
containerId	string	Unique string representing the Container Object to which the process resides. It can be NULL if process isn't in a container.	1
entry	boolean	If true, the process is a container or system entrypoint (i.e., virtual pid = 1).	2
cwd	string	Current working directory of the process.	5
env	string[]	Environment variables array exported to the process.	5

File

The File entity represents file-based resources on a system including files, directories, unix sockets, and pipes.

> **NOTE** Current implementation does not have access to inode related values, which would greatly improve object ids. Also, the current implementation has some issues with absolute paths when monitoring operations that use relative paths.

Attribute	Type	Description	Since (schema version)
state	enum	state of the file (CREATED, MODIFIED, REUP)	1
FOID:	string (128bit)	File Identifier, is a SHA1 hash of the concatenation of the path + container ID	1
timestamp (ts)	int64	The timestamp when file object is exported (nanoseconds).	1
restype	enum	Indicates the resource type. Currently support: SF_FILE, SF_DIR, SF_UNIX (unix socket), SF_PIPE, SF_UNKNOWN	1
path	string	Full path of the file/directory, or unique identifier for pipe, unix socket	1
containerId	string	Unique string representing the Container Object to which the file resides. Can be NULL if file isn't in a container.	1

Events

Events represent important individual behaviors of an entity that are broken out on their own due to their importance, their rarity, or because maintaining operation order is important. In order to manage events and their differing attributes, we divide them into three different categories: Process, File, and Network events. These are described more in detail later on.

Each event and flow contains a process object id, a timestamp, a thread id, and a set of operation flags. The process object id represents the Process Entity on which the event occurred, while the thread id indicates which process thread was associated with the event.

Operation Flags

The operation flags describe the actual behavior associated with the event (or flow). The flags are represented in a single bitmap which enables multiple behaviors to be combined easily into a flow. An event will have a single bit active, while a flow could have several. The current supported flags are as follows:

Operation	Numeric ID	Description	System Calls	Evts/Flows Supported	Since (schema version)
OP_CLONE	(1 << 0)	Process or thread cloned.	clone()	ProcessEvent	1
OP_EXEC	(1 << 1)	Execution of a file	execve()	ProcessEvent	1
OP_EXIT	(1 << 2)	Process or thread exit.	exit()	ProcessEvent	1
OP_SETUID	(1 << 3)	UID of process was changed	setuid(), setresuid	ProcessEvent	1
OP_SETNS	(1 << 4)	Process entering namespace	setns()	FileFlow	1
OP_ACCEP	(1 << 5)	Process accepting network connections	accept(), select()	NetworkFlow	1
OP_CONNE	(1 << 6)	Process connecting to remote host or process	connect()	NetworkFlow	1
OP_OPEN	(1 << 7)	Process opening a file/resource	open(), openat(), create()	FileFlow	1
OP_READ_	(1 << 8)	Process reading from file, receiving network data	read(),pread(),recv(),recv	NetworkFlow, FileFlow	1
OP_WRITE	(1 << 9)	Process writing to file, sending network data	write(),pwrite(),send(),se	NetworkFlow, FileFlow	1
OP_CLOSE	(1 << 10)	Process close resource	close(),socketshutdown	NetworkFlow, FileFlow	1
OP_TRUNC	(1 << 11)	Premature closing of a flow due to exporter shutdown	N/A	NetworkFlow, FileFlow	1
OP_SHUTD	(1 << 12)	Shutdown all or part of a full duplex socket connection	shutdown()	NetworkFlow	1
OP_MMAP	(1 << 13)	Memory map of a file.	mmap()	FileFlow	1
OP_DIGEST	(1 << 14)	Summary flow information for long running flows	N/A	NetworkFlow, FileFlow	1
OP_MKDIR	(1 << 15)	Make directory	mkdir(), mkdirat()	FileEvent	1
OP_RMDIR	(1 << 16)	Remove directory	rmdir()	FileEvent	1
OP_LINK	(1 << 17)	Process creates hard link to existing file	link(), linkat()	FileEvent	1
OP_UNLIN	(1 << 18)	Process deletes file	unlink(), unlinkat()	FileEvent	1
OP_SYMLI	(1 << 19)	Process creates sym link to existing file	symlink(), symlinkat()	FileEvent	1
OP_RENAM	(1 << 20)	File renamed	rename(), renameat()	FileEvent	1

Process Event

A Process Event is an event that creates or modifies a process in some way. Currently, we support four Process Events (referred to as operations), and their behavior in SysFlow is described below.

Operation	Behavior
OP_CLON	Exported when a new process or thread is cloned. A new Process Entity should be exported prior to exporting the clone operation of a new process.
OP_EXEC	Exported when a process calls an exec syscall. This event will modify an existing process, and should be accompanied by a modified Process Entity.
OP_EXIT	Exported on a process or thread exit.
OP_SETU	Exported when a process's UID is changed. This event will modify an existing process, and should be accompanied by a modified Process Entity.

The list of attributes for the Process Event are as follows:

Attribute	Type	Description	Since (schema version)
OID: *host pidcreate ts*	**struct** *int64int64*	The OID of the process for which the event occurred.	1
timestamp (ts)	int64	The timestamp when the event occurred (nanoseconds).	1
tid	int64	The id of the thread associated with the ProcessEvent. If the running process is single threaded tid == pid	1
opFlags	int64	The id of the syscall associated with the event. See list of Operation Flags for details.	1
args	string[]	An array of arguments encoded as string for the syscall.	Sparingly implemented. Only really used with setuid for now.
ret	int64	Syscall return value.	1

File Event

A File Event is an event that creates, deletes or modifies a File Entity. Currently, we support six File Events (referred to as operations), and their behavior in SysFlow is described below.

Operation	Behavior
OP_MKDIR	Exported when a new directory is created. Should be accompanied by a new File Entity representing the directory
OP_RMDIR	Exported when a directory is deleted.
OP_LINK	Exported when a process creates a hard link to an existing file. Should be accompanied by a new File Entity representing the new link.
OP_UNLINK	Exported when a process deletes a file.
OP_SYMLIN	Exported when a process creates a sym link to an existing file. Should be accompanied by a new File Entity representing the new link.
OP_RENAM	Exported when a process creates renames an existing file. Should be accompanied by a new File Entity representing the renamed file.

> **NOTE:** We'd like to also support **chmod** and **chown** but these two operations are not fully supported in sysdig. We'd also like to support **umount** and **mount** but these operations are not implemented. We

anticipate supporting these in a future version.

The list of attributes for the File Event are as follows:

Attribute	Type	Description	Since (schema version)
OID: *host pidcreatts*	**struct** *int64in*	The OID of the process for which the event occurred.	1
timestamp (ts)	int64	The timestamp when the event occurred (nanoseconds).	1
tid	int64	The id of the thread associated with the FileEvent. If the running process is single threaded tid == pid	1
opFlags	int64	The id of the syscall associated with the event. See list of Operation Flags for details.	1
ret	int64	Syscall return value.	1
FOID:	string (128bit	The id of the file on which the system call was called. File Identifier, is a SHA1 hash of the concatenation of the path + container ID.	1
NewFOID:	string (128bit	Some syscalls (link, symlink, etc.) convert one file into another requiring two files. This id is the id of the file secondary or new file on which the system call was called. File Identifier, is a SHA1 hash of the concatenation of the path + container ID. Can be NULL.	1

Network Event

Currently, not implemented.

K8s Event

A k8s Event is an event that records Kubernetes event information.

Attribute	Type	Description	Since (schema version)
timestamp (ts)	int64	The timestamp when the event occurred (nanoseconds).	4
kind	int64	The type of k8s event	1
action	int64	The action associated with the k8s event	4
message	string[]	The detailed k8s event payload or message	4

Flows

A Flow represents an aggregation of multiple events that naturally fit together to describe a particular behavior. They are designed to reduce data and collect statistics. Examples of flows include an application reading or writing to a file, or sending and receiving data from another process or host. Flows represent a number of events occurring over a period of time, and as such each flow has a set of operations (encoded in a bitmap), a start and an end time. One can determine the operations in the flow by decoding the operation flags.

A flow can be started by any supported operation and are exported in one of two ways. First, they are exported on an exit, or close event signifying the end of a connection, file interaction, or process. Second, a long running flow is exported after a preconfigured time period. After a long running flow is exported, its counters and flags are reset. However, if there is no activity on the flow over a preconfigured period of time, that flow is no longer exported.

In this section, we describe three categories of Flows: Process, File and Network Flows.

Process Flow

A Process Flow represents a summarization of the number of threads created and destroyed over a time period. Process Flows are partially implemented in the collector and will be fully implemented in a future release. Since schema version 2. Currently we support the following operations:

Operation	Behavior
OP_CLONE	Recorded when a new thread is cloned.
OP_EXIT	Recorded on a thread exit.

The list of attributes for the Process Flow are as follows:

Attribute	Type	Description	Since (schema version)
OID: *host pidcreate ts*	**struct** *int64int64*	The OID of the process for which the flow occurred.	2
timestamp (ts)	int64	The timestamp when the flow starts (nanoseconds).	2
numThreadsCloned	int64	The number of threads cloned during the duration of the flow.	2
opFlags	int64 (bitmap)	The id of one or more syscalls associated with the ProcessFlow. See list of Operation Flags for details.	2
endTs	int64	The timestamp when the process flow is exported (nanoseconds).	2
numThreadsExited	int64	Number of threads exited during the duration of the flow.	2
numCloneErrors	int64	Number of clone errors occuring during the duration of the flow.	2

File Flow

A File Flow represents a collection of operations on a file. Currently we support the following operations:

Operation	Behavior
OP_SETNS	Process entering namespace entry in mounted file related to reference File Entity
OP_OPEN	Process opening a file/resource.
OP_READ_RECV	Process reading from file/resource.
OP_WRITE_SEND	Process writing to file.
OP_MMAP	Processing memory mapping a file.
OP_CLOSE	Process closing resource. This action will close corresponding FileFlow.
OP_TRUNCATE	Indicates Premature closing of a flow due to exporter shutdown.
OP_DIGEST	Summary flow information for long running flows (not implemented).

The list of attributes for the File Flow are as follows:

Attribute	Type	Description	Since (schema version)
OID: *host pidcreate ts*	**struct** *int64int64*	The OID of the process for which the flow occurred.	1
timestamp (ts)	int64	The timestamp when the flow starts (nanoseconds).	1
tid	int64	The id of the thread associated with the flow. If the running process is single threaded tid == pid	1
opFlags	int64 (bitmap)	The id of one or more syscalls associated with the FileFlow. See list of Operation Flags for details.	1
openFlags	int64	Flags associated with an open syscall if present.	1
endTs	int64	The timestamp when the file flow is exported (nanoseconds).	1
FOID:	string (128bit)	The id of the file on which the system call was called. File Identifier, is a SHA1 hash of the concatenation of the path + container ID.	1
fd	int32	The file descriptor associated with the flow.	1
numRRecvOps	int64	Number of read operations performed during the duration of the flow.	1
numWSendO	int64	Number of write operations performed during the duration of the flow.	1
numRRecvBytes	int64	Number of bytes read during the duration of the flow.	1
numWSendBytes	int64	Number of bytes written during the duration of the flow.	1

Network Flow

A Network Flow represents a collection of operations on a network connection. Currently we support the following operations:

Operation	Behavior
OP_ACCEPT	Process accepted a new network connection.
OP_CONNECT	Process connected to a remote host or process.
OP_READ_RECV	Process receiving data from a remote host or process.
OP_WRITE_SEND	Process sending data to a remote host or process.
OP_SHUTDOWN	Process shutdown full or single duplex connections.
OP_CLOSE	Process closing network connection. This action will close corresponding NetworkFlow.
OP_TRUNCATE	Indicates Premature closing of a flow due to exporter shutdown.
OP_DIGEST	Summary flow information for long running flows (not implemented).

The list of attributes for the Network Flow are as follows:

Attribute	Type	Description	Since (schema version)
OID: *host pidcreate ts*	**struct** *int64int64*	The OID of the process for which the flow occurred.	1
timestamp (ts)	int64	The timestamp when the flow starts (nanoseconds).	1
tid	int64	The id of the thread associated with the flow. If the running process is single threaded tid == pid	1
opFlags	int64 (bitmap)	The id of one or more syscalls associated with the flow. See list of Operation Flags for details.	1
endTs	int64	The timestamp when the flow is exported (nanoseconds).	1
sip	int32	The source IP address.	1
sport	int16	The source port.	1
dip	int32	The destination IP address.	1
dport	int16	The destination port.	1
proto	enum	The network protocol of the flow. Can be: TCP, UDP, ICMP, RAW	1
numRRecvOps	int64	Number of receive operations performed during the duration of the flow.	1
numWSendOps	int64	Number of send operations performed during the duration of the flow.	1
numRRecvBytes	int64	Number of bytes received during the duration of the flow.	1
numWSendBytes	int64	Number of bytes sent during the duration of the flow.	1

NOTE The current implementation of NetworkFlow only supports ipv4.

3.3 LibSysFlow

LibSysFlow is a library for creating SysFlow consumers. It defines a concise API and export first-class SysFlow data types for consumers to transparently process SysFlow records and manage access to the underlying Falco libs and driver.

The main interface accepts a config object in which a callback function can be set to process SysFlow records. The config option sets optimal defaults that can be customized by the consumer. The library is packaged as a static (.a) library and distributed as an rpm/deb/tgz artifact with sf-collector releases (both glibc and musl flavors are available).

Additionally, libsysflow performs the checks to verify that the Falco driver is loaded and outputs an exception otherwise. Consumers load the Falco libs driver prior to running their main entrypoint, following the typical entrypoint recipe/script used by Falco and the SysFlow Collector.

3.3.1 Basic Usage

Below is a minumum example of a SysFlow consumer that uses LibSysFlow. A more complete example can be found here. The SysFlow Collector also uses LibSysFlow and serves as a reference implementation for library consumers.

```
// consumer-defined callback function
void process_sysflow(sysflow::SFHeader* header, sysflow::Container* cont,␣
↪sysflow::Process* proc, sysflow::File* f1, sysflow::File* f2, sysflow::SysFlow* rec) {
    // your switch block here
}

// example consumer
int main(int argc, char **argv) {
    SysFlowConfig* config = sysflowlibscpp::InitializeSysFlowConfig();
    config->callback = process_sysflow;
    sysflowlibscpp::SysFlowDriver *driver = new sysflowlibscpp::SysFlowDriver(config);
    driver->run();
}
```

3.3.2 Public API

The public interface for the SysFlow libs offers two objects: `SysFlowConfig` and `SysFlowDriver`.

SysFlowConfig

The `SysFlowConfig` object is a struct, which contains all settings for the libs and must be passed into the `SysFlowDriver` constructor. A more detailed description of the configuration settings for `SysFlowConfig` can be found in *Advanced Usage*.

Method	Description
SysFlowConfig *sysflowlibscpp::InitializeSysFlowConfig()	Initializes the configuration object with a set of *default* values

SysFlowDriver

The `SysFlowDriver` object is the main object for collecting and exporting SysFlow data. The driver also supports system call ingestion from the following sources: SCAP file, kernel module (live), and ebpf probe (live). Configurations for file ingestion are currently set by the `SysFlowConfig` object. For live capture, the kernel module is loaded by default; however, one can use the ebpf probe by currently exporting the `FALCO_BPF_PROBE` environment variable (e.g., `export FALCO_BPF_PROBE=""`) before launching the binary. Note that probes are launched by running the `falco-driver-loader` script described below. Finally, the driver offers a collection mode option, which determines which system calls are collected. See the `collectionMode` attribute in the *Configuration* section below for more details. The driver currently supports three export options: to avro encoded file, over unix domain socket, and call to user-defined callback function (see example above). Export options are configured using the `SysFlowConfig` option.

Method	Description
SysFlow-Driver(sysflowlibscpp::SysFlowConfig *config	Driver constructor configures the driver based on the settings in the *SysFlowConfig* object.Note: the constructor can throw a *SysFlowException*.
virtual ~SysFlowDriver()	Driver destructor
void exit()	Stops the driver data collection and export. Typically called within a signal handler
int run()	Blocking function that runs the main collection loop. Note: can throw a *SysFlowException*. Returns 0 on successful completion.
std::string getVersion()	returns a string representing the version number of the libraries

3.3.3 Installation

Binary packages (deb, rpm, tgz) for glibc- and musl-based build pipelines are available in the collector's release assets (since release `$VERSION` >=0.5.0). This is going to install libSysFlow headers and the static libraries (.a) needed to link your application.

Debian

Install build pre-requisites:

```
apt install -y make wget g++ libboost-iostreams-dev flex bison gawk libsparsehash-dev
```

Download the libSysFlow package:

```
wget https://github.com/sysflow-telemetry/sf-collector/releases/download/$VERSION/
↪libsysflow-$VERSION-x86_64.deb
```

Install the libSysFlow package:

```
dpkg -i libsysflow-$VERSION-x86_64.deb
```

Note A deb package for musl builds is also available.

RPM

Install pre-requisites (Instructions for Rhel8 below):

```
subscription-manager repos --enable="codeready-builder-for-rhel-8-$(/bin/arch)-rpms"
dnf -y update
dnf -y install make wget gcc gcc-c++ glibc-static libstdc++-static flex bison boost-
↪static sparsehash-devel
```

Download the libSysFlow package:

```
wget https://github.com/sysflow-telemetry/sf-collector/releases/download/$VERSION/
↪libsysflow-$VERSION-x86_64.rpm
```

Install the libSysFlow package:

```
rmp -i libsysflow-$VERSION-x86_64.rpm sfprocessor-$VERSION-x86_64.rpm
```

Note An rpm package for musl builds is also available.

Alpine (musl)

Install pre-requisites:

```
apk add make g++ boost-dev boost-static flex bison gawk sparsehash
```

Download the libSysFlow package:

```
wget https://github.com/sysflow-telemetry/sf-collector/releases/download/$VERSION/
↪libsysflow-$VERSION-x86_64.tgz
```

Install the libSysFlow package:

```
tar xzf libsysflow-musl-${SYSFLOW_VERSION}-x86_64.tar.gz && cp -r libsysflow-musl-$
↪{SYSFLOW_VERSION}-x86_64/usr/* /usr/.
```

3.3.4 Compilation

After installation, you should have the following directory structure installed in your environment:

```
/usr/bin/docker-entrypoint.sh
/usr/bin/falco-driver-loader
/usr/include/falcosecurity
/usr/include/sysflow
/usr/lib/falcosecurity
/usr/lib/sysflow
/usr/src/dkms
/usr/src/falco-$FALCO_LIBS_VERSION
```

The `include` and `lib` directories contain the header files and static libraries that should be used to build a SysFlow consumer. `docker-entrypoint.sh` is provided for container-based deployments of the consumer, and the `falco-driver-loader` is the script used to load the kmod/bpf drivers. `dkms` sources are provided as a convenience and not required for compilation, and can be used to install dkms in case it's not available in the target environment. Similarly, we package the Falco driver sources in `/usr/src/falco-$FALCO_LIBS_VERSION` to enable local compilation of the drivers when these are not available or accessible in the remote driver repository.

The Makefile in the example application shows the `LDFLAGS` and `CFLAGS` needed to build a libSysFlow consumer, and provides an example of how to enable glibc and musl (static) builds.

3.3.5 Advanced Usage

Configuration

The library configuration parameters are assigned defaults that should work well in most scenarios. They can be customized using the `SysFlowConfig` object.

Field	Type	Description	Default
filterContainers	bool	Filter out all events related to containers	false
rotateInterval	int	Set rotation interval in secs which dictates how often a SysFlow header is emitted. Used for file rotations and also to clean caches to prevent leakages	300
exporterID	string	ID for the host	
nodeIP	string	IP for the host/node	
filePath	string	SysFlow output file path. If path ends with a '/', this will be treated as a directory. If treated as directory, the name of the sysflow file will be a timestamp, and will be rotated every N seconds depending on the rotateInterval. If no '/' at end, and rotateInterval is set, path is treated as a file prefix, and timestamp is concatenated. Set NULL if not using file output.	
socketPath	string	SysFlow unix socket file path. Typically used in conjunction with the SysFlow processor to stream SysFlow over a socket. Set NULL if not using socket streaming	
scapInputPath	string	Scap input file path. Used in offline mode to read from raw scap rather than tapping the kernel. Set NULL if using live kernel collection	
falcoFilter	string	String to set Falco-style filter on events being passed from the falco libs, to the SysFlow library	
samplingRatio	string	Sampling ratio used to determine which system calls to drop in the probe	1
criPath	string	CRI-O runtime socket path, needed for monitoring cri-o/containered container runtimes such as k8s and OCP	
criTO	int	CRI-O timeout. Timeout in secs set when querying CRI-O socket for container metadata	30
enableSta	bool	Enable Process Flow collection. Output Process Flow rather than individual thread clones	false
enablePrc ess-Flow	bool	Only output File Flows and Events related to file objects. Ignoring pipes, for example.	true
fileOnl	bool	Only output File Flows and Events related to file objects. Ignoring pipes, for example.	true
fileReadMode	int	Set the file mode to determine which types of file related read flows are ignored to reduce event output. sets mode for reads: "0" enables recording all file reads as flows. "1" disables all file reads. "2" disables recording file reads to noisy directories: "/proc/", "/dev/", "/sys/", "//sys/", "/lib/", "/lib64/", "/usr/lib/", "/usr/lib64/"	2
dropMode	bool	Drop mode removes syscalls inside the kernel before they are passed up to the collector results in much better performance, less drops, but does remove mmaps from output.	true
callback	SysFlowCallback	Callback function, required for when using a custom callback function for SysFlow processing	
debugMode	bool	Debug mode turns on debug logging inside libsinsp	false
k8sAPI	string	K8s API URL used to retrieve K8s state and K8s events (experimental)	
k8sAPI	string	Path to K8s API Certificate (experimental)	
moduleChe	bool	Run added module checks for better error checking	true
col-	enum	Has three possible values: 1.) `SFFlowMode` for SysFlow mode which does full SysFlow	`SFFlowMode`

Exception Handling

The library exposes an exception class that contains error code that can be used by SysFlow consumers for logging and troubleshooting.

```
SysFlowException(std::string message);
SysFlowException(std::string message, SysFlowError code)
      : std::runtime_error(message), m_code(code) {}
SysFlowError getErrorCode() { return m_code; }
```

The error codes are defined in an enum, as follows.

```
enum SysFlowError {
  LibsError,
  ProbeAccessDenied,
  ProbeNotExist,
  ErrorReadingFileSystem,
  NameTooLong,
  ProbeCheckError,
  ProbeNotLoaded,
  DriverLibsMismatch,
  EventParsingError,
  ProcResourceNotFound,
  OperationNotSupported
};
```

Logging

LibSysFlow uses Glog for logging.

You can specify one of the following severity levels (in increasing order of severity): INFO, WARNING, ERROR.

You can also add verbose logging when you are chasing difficult bugs. LibSysFlow has two levels of verbose logging: 1 (DEBUG) and 2 (TRACE).

To control logging behavior, you can set flags via environment variables, prefixing the flag name with "GLOG", e.g.

```
GLOG_logtostderr=1 ./your_application
```

The following flags are most commonly used:

- logtostderr (bool, default=false): Log messages to stderr instead of logfiles.
- stderrthreshold (int, default=2, which is ERROR): Copy log messages at or above this level to stderr in addition to logfiles. The numbers of severity levels INFO, WARNING, ERROR, and FATAL are 0, 1, 2, and 3, respectively.
- minloglevel (int, default=0, which is INFO): Log messages at or above this level. Again, the numbers of severity levels INFO, WARNING, ERROR, and FATAL are 0, 1, 2, and 3, respectively.
- log_dir (string, default=""): If specified, logfiles are written into this directory instead of the default logging directory.
- v (int, default=0): Show all VLOG(m) messages for m less or equal the value of this flag. Overridable by –vmodule. See the section about verbose logging for more detail.
- vmodule (string, default=""): Per-module verbose level. The argument has to contain a comma-separated list of =. is a glob pattern (e.g., gfs* for all modules whose name starts with "gfs"), matched against the filename base (that is, name ignoring .cc/.h./-inl.h). overrides any value given by –v.

Note You can set binary flags to true by specifying 1, true, or yes (case insensitive). Also, you can set binary flags to false by specifying 0, false, or no (again, case insensitive).

3.4 SysFlow Collector (sf-collector repo)

The SysFlow Collector monitors and collects system call and event information from hosts and exports them in the SysFlow format using Apache Avro object serialization. It's built atop libSysFlow, a library that lifts system call information into SysFlow, a higher order object relational format that models how containers, processes and files interact with their environment through process control flow, file, and network operations. Learn more about SysFlow in the SysFlow Specification Document.

The SysFlow Collector builds on the CNCF Falco libs to passively collect system events and turn them into SysFlow. As a result, the collector supports the libs' powerful filtering capabilities. Check the build and installation instructions for installing the collector.

3.4.1 Build

Cloning sources

This document describes how to build libSysFlow and run the SysFlow Collector both inside a docker container and on a linux host. Binary packages are also available in the deployments repository. Building and running the application inside a docker container is the easiest way to start. For convenience, skip the build step and pull pre-built images directly from Docker Hub.

To build the project, first clone the repository:

```
git clone --recursive https://github.com/sysflow-telemetry/sf-collector.git
```

Manifest

The manifest file contains the metadata and versions of dependencies used to build libSysFlow and the Collector. It can be modified to customize the build to specifc package versions.

Building using Docker

This is the simplest way of reliably building the collector. To build using docker, run:

```
make build
```

Note A musl build can be triggered using the `build/musl` target instead.

If this is your first time building the collector, run the build task in the background and go grab a coffee :) If you have cores to spare, the build time can be reduced by setting concurrent make jobs. For example,

```
make MAKE_JOBS=8 build
```

During the initial build, a number of base images are created. These are only needed once per dependency version set. Pre-built versions of these images are also available in Docker Hub and GHCR.

Image	Tag	Description	Dockerfile
ghcr.io/sysflow-telemetry/ubi	base--	A UBI base image containing the build dependencies for Falco and libSysFlow	Docker-file.ubi.amd64
	mods--	A UBI base image containing the pre-installed Falco Libs and tools for building libSysFlow	Docker-file.ubi.amd64
	driver--	A UBI base image containing the Falco driver loader and container entrypoint for creating the SysFlow Collector released image	Docker-file.driver.amd64
ghcr.io/sysflow-telemetry/sf-collector	libs-	A UBI base image containing libSysFlow	Dockerfile
	collect	A UBI base image containing the SysFlow Collector	Dockerfile
		The SysFlow Collector image	Dockerfile

If building using musl, the following images are created instead.

Image	Tag	Description	Dockerfile
ghcr.io/sysflow-telemetry/alpine	base--	An Alpine base image containing the musl build dependencies for Falco and libSysFlow	Docker-file.alpine.amd64
	mods--	An Alpine base image containing the pre-installed Falco Libs and tools for building a musl-based libSysFlow	Docker-file.alpine.amd64
ghcr.io/sysflow-telemetry/ubi	driver--	A UBI base image containing the Falco driver loader and container entrypoint for creating the SysFlow Collector released image	Docker-file.driver.amd64
ghcr.io/sysflow-telemetry/sf-collector-musl	libs-	An Alpine base image containing musl-based libSysFlow	Docker-file.musl
	collect	An Alpine base image containing the musl-based SysFlow Collector	Docker-file.musl
		The musl-based SysFlow Collector image	Docker-file.musl

Building directly on a host

First, install required dependencies.

On Rhel-based hosts:

```
scripts/installUBIDependency.sh
```

On Debian-based hosts:

```
apt install -y patch base-files binutils bzip2 libdpkg-perl perl make xz-utils␣
↪libncurses5-dev libncursesw5-dev cmake libboost-all-dev g++ flex bison wget libelf-dev␣
↪liblog4cxx-dev libapr1 libaprutil1 libsparsehash-dev libsnappy-dev libgoogle-glog-dev␣
↪libjsoncpp-dev
```

To build the collector:

```
make
```

Binary Packaging

You can easily package libSysFlow and the Collector using `cpack`. The deb, rpm, and tgz packages are generated in the `sctrips/cpack` directory.

To package libSysFlow headers and static libraries, run:

```
make build
make package-libs
```

To package a musl-based libSysFlow headers and static libraries, run:

```
make build/musl
make package-libs/musl
```

To package the SysFlow collector and its systemd service specification, run:

```
make build/musl
make package
```

3.4.2 Running

Command line usage

To list command line options for the collector, run:

```
sysporter -h
```

Examples

To convert `scap` files to SysFlow traces with an export id. The output will be written to `output.sf`.

```
sysporter -r input.scap -w ./output.sf  -e host
```

Trace a system live, and output SysFlow to files in a directory which are rotated every 30 seconds. The file name will be an epoch timestamp of when the file was initially written. Note that the trailing slash *must be present*. The example filter ensures that only SysFlow from containers is generated.

```
sysporter -G 30 -w ./output/ -e host -f "container.type!=host and container.type=docker"
```

Trace a system live, and output SysFlow to files in a directory which are rotated every 30 seconds. The file name will be an `output.<epoch timestamp>` where the timestamp is of when the file was initially written. The example filter ensures that only SysFlow from containers is generated.

```
sysporter -G 30 -w ./output/output -e host -f "container.type!=host and container.
↪type=docker" </code>`
```

Docker usage

The easiest way to run the SysFlow collector is from a Docker container, with host mount for the output trace files. The following command shows how to run sf-collector with trace files exported to `/mnt/data` on the host.

```
docker run -d --privileged --name sf-collector \
           -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 \
           -v /var/lib/:/host/var/lib:ro \
           -v /mnt/data:/mnt/data \
           -e INTERVAL=60 \
           -e EXPORTER_ID=${HOSTNAME} \
           -e OUTPUT=/mnt/data/    \
           -e FILTER="container.name!=sf-collector and container.name!=sf-processor and␣
↪container.name!=sf-exporter" \
           --rm sysflowtelemetry/sf-collector
```

where `INTERVAL` denotes the time in seconds before a new trace file is generated, `EXPORTER_ID` sets the exporter name, `OUTPUT` is the directory in which trace files are written, and `FILTER` is the filter expression used to filter collected events. The collector can also be setup to stream events through a unix domain socket (see sf-deployments for other deployment configurations).

> **Note** append `container.type!=host` to FILTER expression to filter host events.

The key setting in the collector configuration is the `FILTER` variable. The collector is built atop the Falco libs and it uses Falco's filtering mechanism described here. It supports filtering on specific containers, processes, operations, etc. One of the most powerful filters is the `container.type!=host` filter, which limits collection only to container monitoring. If you want to monitor the entire host, simply remove the `container.type` operation from the filter.

3.4.3 Event rate optimization

The following environment variables can be set to reduce the number of events generated by the collector:

- Drop mode (`ENABLE_DROP_MODE=1`): removes syscalls inside the kernel before they are passed up to the collector, resulting in much better performance, less spilled events, but does remove mmaps from output.
- Process flows (`ENABLE_PROC_FLOW=1`): enables the creation of process flows, aggregating thread events.
- File only (`FILE_ONLY=1`): filters out any descriptor that is not a file, including unix sockets and pipes
- File read mode (`FILE_READ_MODE=1`): sets mode for file reads. `0` enables recording all file reads as flows. `1` disables all file reads. `2` disables recording file reads to noisy directories: "/proc/", "/dev/", "/sys/", "//sys/", "/lib/", "/lib64/", "/usr/lib/", "/usr/lib64/".

3.5 SysFlow Processor (sf-processor repo)

The SysFlow processor is a lighweight edge analytics pipeline that can process and enrich SysFlow data. The processor is written in golang, and allows users to build and configure various pipelines using a set of built-in and custom plugins and drivers. Pipeline plugins are producer-consumer objects that follow an interface and pass data to one another through pre-defined channels in a multi-threaded environment. By contrast, a driver represents a data source, which pushes data to the plugins. The processor currently supports two builtin drivers, including one that reads sysflow from a file, and another that reads streaming sysflow over a domain socket. Plugins and drivers are configured using a JSON file.

A core built-in plugin is a policy engine that can apply logical rules to filter, alert, or semantically label sysflow records using a declarative language based on the Falco rules syntax with a few added extensions (more on this later).

Custom plugins and drivers can be implemented as dynamic libraries to tailor analytics to specific user requirements.

The endpoint of a pipeline configuration is an exporter plugin that sends the processed data to a target. The processor supports various types of export plugins for a variety of different targets.

3.5.1 Pre-requisites

The processor has been tested on Ubuntu/RHEL distributions, but should work on any Linux system.

- Golang version 1.17+ and make (if building from sources)
- Docker, docker-compose (if building with docker)

3.5.2 Build

Clone the processor repository

```
git clone https://github.com/sysflow-telemetry/sf-processor.git
```

Build locally, from sources

```
cd sf-processor
make build
```

Build with docker

```
cd sf-processor
make docker-build
```

3.5.3 Usage

For usage information, type:

```
cd driver/
./sfprocessor -help
```

This should yield the following usage statement:

```
Usage: sfprocessor [[-version]|[-driver <value>] [-log <value>] [-driverdir <value>] [-
↪plugdir <value>] path]
Positional arguments:
  path string
        Input path
Arguments:
  -config string
        Path to pipeline configuration file (default "pipeline.json")
  -cpuprofile file
        Write cpu profile to file
  -driver string
        Driver name {file|socket|<custom>} (default "file")
  -driverdir string
        Dynamic driver directory (default "../resources/drivers")
  -log string
        Log level {trace|info|warn|error} (default "info")
  -memprofile file
        Write memory profile to file
  -plugdir string
        Dynamic plugins directory (default "../resources/plugins")
  -test
        Test pipeline configuration
  -traceprofile file
        Write trace profile to file
  -version
        Output version information
```

The four most important flags are `config`, `driverdir`, `plugdir`, and `driver`. The `config` flag points to a pipeline configuration file, which describes the entire pipeline and settings for the individual settings for the plugins. The `driverdir` and `plugdir` flags specify where any dynamic drivers and plugins shared libraries reside that should be loaded by the processor at runtime. The `driver` flag accepts a label to a pre-configured driver (either built-in or custom) that will be used as the data source to the pipeline. Currently, the pipeline only supports one driver at a time, but we anticipate handling multiple drivers in the future. There are two built-in drivers:

- *file*: loads a sysflow file reading driver that reads from `path`.
- *socket*: the processor loads a sysflow streaming driver. The driver creates a domain socket named `path` and acts as a server waiting for a SysFlow collector to attach and send sysflow data.

3.5.4 Configuration

The pipeline configuration below shows how to configure a pipeline that will read a sysflow stream and push records to the policy engine, which will trigger alerts using a set of runtime policies stored in a `yaml` file. An example pipeline with this configuration looks as follows:

```
{
  "pipeline":[
    {
     "processor": "sysflowreader",
     "handler": "flattener",
     "in": "sysflow sysflowchan",
     "out": "flat flattenerchan"
    },
```

(continues on next page)

(continued from previous page)

```
    {
     "processor": "policyengine",
     "in": "flat flattenerchan",
     "out": "evt eventchan",
     "policies": "../resources/policies/runtimeintegrity"
    },
    {
     "processor": "exporter",
     "in": "evt eventchan",
     "export": "syslog",
     "proto": "tcp",
     "tag": "sysflow",
     "host": "localhost",
     "port": "514"
    }
  ]
}
```

NOTE: This configuration can be found in: `sf-collector/resources/pipelines/pipeline.syslog.json`

This pipeline specifies three built-in plugins:

- sysflowreader: is a generic reader plugin that ingests sysflow from the driver, caches entities, and presents sysflow objects to a handler object (i.e., an object that implements the handler interface) for processing. In this case, we are using the flattener handler, but custom handlers are possible.
- policyengine: is the policy engine, which takes flattened (row-oriented) SysFlow records as input and outputs records, which represent alerts, or filtered sysflow records depending on the policy engine's *mode* (more on this later).
- exporter: takes records from the policy engine, and exports them to ElasticSearch, syslog, file, or terminal, in a JSON format or in Elastic Common Schema (ECS) format. Note that custom export plugins can be created to export to other serialization formats and transport protocols.

Each plugin has a set of general attributes that are present in all plugins, and a set of attributes that are custom to the specific plugins. For more details on the specific attributes in this example, see the pipeline configuration template

The general attributes are as follows:

- *processor* (required): the name of the processor plugin to load. Processors must implement the SFProcessor interface; the name is the value that must be returned from the `GetName()` function as defined in the processor object.
- *handler* (optional): the name of the handler object to be used for the processor. Handlers must implement the SFHandler interface.
- *in* (required): the input channel (i.e. golang channel) of objects that are passed to the plugin.
- *out* (optional): the output channel (i.e. golang channel) for objects that are pushed out of the plugin, and into the next plugin in the pipeline sequence.

Channels are modelled as channel objects that have an `In` attribute representing some golang channel of objects. See SFChannel for an example. The syntax for a channel in the pipeline is `[channel name] [channel type]`. Where channel type is the label given to the channel type at plugin registration (more on this later), and channel name is a unique identifier for the current channel instance. The name and type of an output channel in one plugin must match that of the name and type of the input channel of the next plugin in the pipeline sequence.

NOTE: A plugin has exacly one input channel but it may specify more than one output channels. This allows pipeline definitions that fan out data to more than one receiver plugin similar to a Unix `tee` command. While there must be always one SysFlow reader acting as the entry point of a pipeline, a pipeline configuration may specify policy engines passing data to different exporters or a SysFlow reader passing data to different policy engines. Generally, pipelines form a tree rather being a linear structure.

Policy engine configuration

The policy engine (`"processor": "policyengine"`) plugin is driven by a set of rules. These rules are specified in a YAML file which adopts the same syntax as the rules of the Falco project. A policy engine plugin specification may have the following attributes:

- *policies* (required for `alert` mode`): The path to the YAML rules specification file. More information on rules can be found in the Policies section.
- *mode* (optional): The mode of the policy engine. Allowed values are:
  - `alert` (default): the policy engine generates rule-based alerts; `alert` is a blocking mode that drops all records that do not match any given rule. If no mode is specified, the policy engine runs in `alert` mode by default.
  - `enrich` for enriching records with additional context from the rule. In contrast to `alert`, this is a non-blocking mode which applies tagging and action enrichments to matching records as defined in the policy file. Non-matching records are passed on "as is".
- *monitor* (optional): Specifies if changes to the policy file(s) should be monitored and updated in the policy engine.
  - `none` (default): no monitor is used.
  - `local`: the processor will monitor for changes in the policies path and update its rule set if changes are detected.
- *monitor.interval* (optional): The interval in seconds for updating policies, if a monitor is used. (default: 30 seconds).
- *concurrency* (optional); The number of concurrent threads for record processing. (default: 5).
- *actiondir* (optional): The path of the directory containing the shared object files for user-defined action plugins. See the section on User-defined Actions for more information.

  NOTE: Prior to release 0.4.0, the *mode* attribute accepted different values with different semantics. To preserve the behavior of older releases:
  - For old `alert` behavior, use `enrich` mode.
  - For old `filter` behavior, use `enrich` mode and a policy file with filter rules only.
  - For old `bypass` behavior, use `enrich` and drop the *policies* key from the configuration.

Exporter configuration

An exporter (`"processor": "exporter"`) plugin consists of two modules, an encoder for converting the data to a suitable format, and a transport module for sending the data to the target. Encoders target specific, i.e. for a particular export target a particular set of encoders may be used. In the exporter configuration the transport module is specified via the *export* parameter (required). The encoder is selected via the *format* parameter (optional). The default format is `json`.

The following table lists the currently supported exporter modules and the corresponding encoders. Additional encoders and transport modules can be implemented if need arises. If you plan to contribute or want to get involved in the discussion please join the SysFlow community.

Some of these combinations require additional configuration as described in the following sections. `null` is used for debugging the processor and doesn't export any data.

File

If *export* is set to `file`, an additional parameter *file.path* allows the specification of the target file.

Syslog

If the *export* parameter is set to `syslog`, output to syslog is enabled and the following addtional parameters are used:

- *syslog.proto* (optional): The protocol used for communicating with the syslog server. Allows values are `tcp`, `tls` and `udp`. Default is `tcp`.
- *syslog.tag* (optional): The tag used for each Sysflow record in syslog. Default is `SysFlow`.
- *syslog.source* (optional): If set adds a hostname to the syslog header.
- *syslog.host* (optional): The hostname of the sysflow server. Default is `localhost`.
- *syslog.port* (optional): The port of the syslow server. Default is `514`.

ElasticSearch

Export to ElasticSearch is enabled by setting the config parameter *export* to `es`. The only supported *format* for export to ElasticSearch is `ecs`.

Data export is done via bulk ingestion. The ingestion can be controlled by some additional parameters which are read when the `es` export target is selected. Required parameters specify the ES target, index and credentials. Optional parameters control some aspects of the behavior of the bulk ingestion and may have an effect on performance. You may need to adapt their valuesfor optimal performance in your environment.

- *es.addresses* (required): A comma-separated list of ES endpoints.
- *es.index* (required): The name of the ES index to ingest into.
- *es.username* (required): The ES username.
- *es.password* (required): The password for the specified ES user.
- *buffer* (optional) The bulk size as the number of records to be ingested at once. Default is `0` but value of `0` indicates record-by-record ingestion which may be highly inefficient.
- *es.bulk.numWorkers* (optional): The number of ingestion workers used in parallel. Default is `0` which means that the exporter uses as many workers as there are cores in the machine.

- *es.bulk.flashBuffer* (optional): The size in bytes of the flush buffer for ingestion. It should be large enough to hold one bulk (the number of records specified in *buffer*), otherwise the bulk is broken into smaller chunks. Default is `5e+6`.
- *es.bulk.flushTimeout* (optional): The flush buffer time threshold. Valid values are golang duration strings. Default is `30s`.

The Elastic exporter does not make any assumption on the existence or configuration of the index specified in *es.index*. If the index does not exist, Elastic will automatically create it and apply a default dynamic mapping. It may be beneficial to use an explicit mapping for the ECS data generated by the Elastic exporter. For convinience we provide an explicit mapping for creating a new tailored index in Elastic. For more information refer to the Elastic Mapping reference.

Environment variables

It is possible to override any of the custom attributes of a plugin using an environment variable. This is especially useful when operating the processor as a container, where you may have to deploy the processor to multiple nodes, and have attributes that change per node. If an environment variable is set, it overrides the setting inside the config file. The environment variables must follow the following structure:

- Environment variables must follow the naming schema `<PLUGIN NAME>_<CONFIG ATTRIBUTE NAME>`
- The plugin name inside the pipeline configuration file must be all lower case.

For example, to set the alert mode inside the policy engine, the following environment variable is set:

```
export POLICYENGINE_MODE=alert
```

To set the syslog values for the exporter:

```
export EXPORTER_TYPE=telemetry
export EXPORTER_SOURCE=${HOSTNAME}
export EXPORTER_EXPORT=syslog
export EXPORTER_HOST=192.168.2.10
export EXPORTER_PORT=514
```

If running as a docker container, environment variables can be passed with the docker run command:

```
docker run
-e EXPORTER_TYPE=telemetry \
-e EXPORTER_SOURCE=${HOSTNAME} \
-e EXPORTER_EXPORT=syslog \
-e EXPORTER_HOST=192.168.2.10 \
-e EXPORTER_PORT=514
...
```

Rate limiter configuration (experimental)

The `flattener` handler has a built-in time decay filter that can be enabled to reduce even rates in the processor. The filter uses a time-decay bloom filter based on a semantic hashing of records. This means that the filter should only forward one record matching a semantic hash per time decay period. The semantic hash takes into consideration process, flow and event attributes. To enable rate limiting, modify the `sysflowreader` processor as follows:

```
{
     "processor": "sysflowreader",
     "handler": "flattener",
```

(continues on next page)

(continued from previous page)

```
    "in": "sysflow sysflowchan",
    "out": "flat flattenerchan",
    "filter.enabled": "on|off (default: off)",
    "filter.maxage": "time decay in minutes (default: 24H)"
}
```

3.5.5 Policy Language

The policy engine adopts and extends the Falco rules definition syntax. Before reading the rest of this section, please go through the Falco Rules documentation to get familiar with *rule*, *macro*, and *list* syntax, all of which are supported in our policy engine. Policies are written in one or more `yaml` files, and stored in a directory specified in the pipeline configuration file under the `policies` attribute of the policy engine plugin.

Rules contain the following fields:

- *rule*: the name of the rule
- *description*: a textual description of the rule
- *condition*: a set of logical operations that can reference lists and macros, which when evaluating to *true*, can trigger record enrichment or alert creation (depending on the policy engine mode)
- *action*: a comma-separated list of actions to take place when the rule evaluates to *true*. For a particular rule, actions are evaluated in the order they are specified, i.e., an action can make use of the results provided by earlier actions. An action is just the name of an action function without any parameters. The current version only supports plugable user-defined actions. See *here* for a detailed description of the plugin interface and a sample action plugin.
- *priority*: label representing the severity of the alert can be: (1) low, medium, or high, or (2) emergency, alert, critical, error, warning, notice, informational, debug.
- *tags* (optional): set of labels appended to alert (default: empty).
- *prefilter* (optional): list of record types (`sf.type`) to whitelist before applying rule condition (default: empty).
- *enabled* (optional): indicates whether the rule is enabled (default: true).

  NOTE: The syntax of the policy language changed slighly with the switch to release 0.4.0. For migrating policy files used with prior releases to release 0.4.0 or higher, simply remove all `action: [tag]` lines. As of release 0.4.0, tagging is done automatically. If a rule triggers all tags specified via the *tags* key will be appended to the record. The *action* key is reserved for specifying user-defined action plugins.</p>

Macros are named conditions and contain the following fields:

- *macro*: the name of the macro
- *condition*: a set of logical operations that can reference lists and macros, which evaluate to *true* or *false*

Lists are named collections and contain the following fields:

- *list*: the name of the list
- *items*: a collection of values or lists

Drop rules block records matching a condition and can be used for reducing the amount of records processed by the policy engine:

- *drop*: the name of the filter
- *condition*: a set of logical operations that can reference lists and macros, which evaluate to *true* or *false*

For example, the rule below specifies that matching records are process events (`sf.type = PE`), denoting `EXEC` operations (`sf.opflags = EXEC`) for which the process matches macro `package_installers`. Additionally, the global filter `containers` preemptively removes from the processing stream any records for processes not running in a container environment.

```
# lists
- list: rpm_binaries
  items: [dnf, rpm, rpmkey, yum, '"75-system-updat"', rhsmcertd-worke, subscription-ma,
          repoquery, rpmkeys, rpmq, yum-cron, yum-config-mana, yum-debug-dump,
          abrt-action-sav, rpmdb_stat, microdnf, rhn_check, yumdb]

- list: deb_binaries
  items: [dpkg, dpkg-preconfigu, dpkg-reconfigur, dpkg-divert, apt, apt-get, aptitude,
    frontend, preinst, add-apt-reposit, apt-auto-remova, apt-key,
    apt-listchanges, unattended-upgr, apt-add-reposit]

- list: package_mgmt_binaries
  items: [rpm_binaries, deb_binaries, update-alternat, gem, pip, pip3, sane-utils.post,␣
↪alternatives, chef-client]

# macros
- macro: package_installers
  condition: sf.proc.name pmatch (package_mgmt_binaries)

# global filters (blacklisting)
- filter: containers
  condition: sf.container.type = host

# rule definitions
- rule: Package installer detected
  desc: Use of package installer detected
  condition:  sf.opflags = EXEC and package_installers
  priority: medium
  tags: [actionable-offense, suspicious-process]
  prefilter: [PE] # record types for which this rule should be applied (whitelisting)
  enabled: true
```

Attribute names

The following table shows a detailed list of attribute names supported by the policy engine, as well as their type, and comparative Falco attribute name. Our policy engine supports both SysFlow and Falco attribute naming convention to enable reuse of policies across the two frameworks.

Attributes	Description	Values
sf.type	Record type	PE,PF,NF,FF,FE,KE
sf.opflags	Operation flags	Operation Flags List: remove OP_ prefix
sf.ret	Return code	int
sf.ts	start timestamp(ns)	int64
sf.endts	end timestamp(ns)	int64
sf.proc.pid	Process PID	int64
sf.proc.tid	Thread PID	int64

Attributes	Description	Values
sf.proc.uid	Process user ID	int
sf.proc.user	Process user name	string
sf.proc.gid	Process group ID	int
sf.proc.group	Process group name	string
sf.proc.apid	Proc ancestors PIDs (qo)	int64
sf.proc.aname	Proc anctrs names (qo) (exclude path)	string
sf.proc.exe	Process command/filename (with path)	string
sf.proc.args	Process command arguments	string
sf.proc.name	Process name (qo) (exclude path)	string
sf.proc.cmdline	Process command line (qo)	string
sf.proc.tty	Process TTY status	boolean
sf.proc.entry	Process container entrypoint	bool
sf.proc.createts	Process creation timestamp (ns)	int64
sf.pproc.pid	Parent process ID	int64
sf.pproc.gid	Parent process group ID	int64
sf.pproc.uid	Parent process user ID	int64
sf.pproc.group	Parent process group name	string
sf.pproc.tty	Parent process TTY status	bool
sf.pproc.entry	Parent process container entry	bool
sf.pproc.user	Parent process user name	string
sf.pproc.exe	Parent process command/filename	string
sf.pproc.args	Parent process command arguments	string
sf.pproc.name	Parent process name (qo) (no path)	string
sf.pproc.cmdline	Parent process command line (qo)	string
sf.pproc.createts	Parent process creation timestamp	int64
sf.file.fd	File descriptor number	int
sf.file.path	File path	string
sf.file.newpath	New file path (used in some FileEvents)	string
sf.file.name	File name (qo)	string
sf.file.directory	File directory (qo)	string
sf.file.type	File type	char ‘f’: file, 4: IPv4, 6: IPv6, ‘u’: unix socket, ‘p’: pipe, ‘e’: ever
sf.file.is_open_write	File open with write flag (qo)	bool
sf.file.is_open_read	File open with read flag (qo)	bool
sf.file.openflags	File open flags	int
sf.net.proto	Network protocol	int
sf.net.sport	Source port	int
sf.net.dport	Destination port	int
sf.net.port	Src or Dst port (qo)	int
sf.net.sip	Source IP	int
sf.net.dip	Destination IP	int
sf.net.ip	Src or dst IP (qo)	int
sf.res	File or network resource	string
sf.flow.rbytes	Flow bytes read/received	int64
sf.flow.rops	Flow operations read/received	int64
sf.flow.wbytes	Flow bytes written/sent	int64
sf.flow.wops	Flow bytes written/sent	int64
sf.container.id	Container ID	string
sf.container.name	Container name	string
sf.container.image.id	Container image ID	string
sf.container.image	Container image name	string

Attributes	Description	Values
sf.container.type	Container type	CT_DOCKER, CT_LXC, CT_LIBVIRT_LXC, CT_MESOS, CT_
sf.container.privileged	Container privilege status	bool
sf.pod.ts	Pod creation timestamp	int
sf.pod.id	Pod id	string
sf.pod.name	Pod name	string
sf.pod.nodename	Pod node name	string
sf.pod.namespace	Pod namespace	string
sf.pod.restartcnt	Pod restart count	int
sf.pod.hostip	Pod host IP addresses	json
sf.pod.internalip	Pod internal IP addresses	json
sf.pod.services	Pod services	json
sf.ke.action	Kubernetes event action	K8S_COMPONENT_ADDED, K8S_COMPONENT_MODIFIE
sf.ke.kind	Kubernetes event resource type	K8S_NODES, K8S_NAMESPACES, K8S_PODS, K8S_REPLIC
sf.ke.message	Kubernetes event json message	json
sf.node.id	Node identifier	string
sf.node.ip	Node IP address	string
sf.schema.version	SysFlow schema version	string
sf.version	SysFlow JSON schema version	int

$ Jsonpath Expressions

Unlike attributes of the scalar types bool, int(64), and string, attributes of type `json` contain structured information in form of stringified json records. The policy language allows access to subfields inside such json records via GJSON jsonpath expressions. The jsonpath expression must be specified as a suffix to the attribute enclosed in square brackets. Examples of such terms are:

```
sf.pod.services[0.clusterip.0]   - the first cluster IP address of the first service␣
↪associated with  a pod
sf.ke.message[items.0.namespace] - the namespace of the first item in a KE message␣
↪attribute
```

See the GJSON path synax for more details. The result of applying a jsonpath expression to a json attribute is always of type string.

Operations

The policy language supports the following operations:

Operation	Description	Example
A and B	Returns true if both statements are true	sf.pproc.name=bash and sf.pproc.cmdline contains echo
A or B	Returns true if one of the statements are true	sf.file.path = "/etc/passwd" or sf.file.path = "/etc/shadow"
not A	Returns true if the statement isn't true	not sf.pproc.exe = /usr/local/sbin/runc
A = B	Returns true if A exactly matches B. Note, if B is a list, A only has to exact match one element of the list. If B is a list, it must be explicit. It cannot be a variable. If B is a variable use `in` instead.	sf.file.path = ["/etc/passwd", "/etc/shadow"]
A != B	Returns true if A is not equal to B. Note, if B is a list, A only has to be not equal to one element of the list. If B is a list, it must be explicit. It cannot be a variable.	sf.file.path != "/etc/passwd"
A < B	Returns true if A is less than B. Note, if B is a list, A only has to be less than one element in the list. If B is a list, it must be explicit. It cannot be a variable.	sf.flow.wops < 1000
A <= B	Returns true if A is less than or equal to B. Note, if B is a list, A only has to be less than or equal to one element in the list. If B is a list, it must be explicit. It cannot be a variable.	sf.flow.wops <= 1000
A > B	Returns true if A is greater than B. Note, if B is a list, A only has to be greater than one element in the list. If B is a list, it must be explicit. It cannot be a variable.	sf.flow.wops > 1000
A >= B	Returns true if A is greater than or equal to B. Note, if B is a list, A only has to be greater than or equal to one element in the list. If B is a list, it must be explicit. It cannot be a variable.	sf.flow.wops >= 1000
A in B	Returns true if value A is an exact match to one of the elements in list B. Note: B must be a list. Note: () can be used on B to merge multiple list objects into one list.	sf.proc.exe in (bin_binaries, usr_bin_binaries)
A startsw B	Returns true if string A starts with string B	sf.file.path startswith '/home'
A endswi B	Returns true if string A ends with string B	sf.file.path endswith '.json'
A contains B	Returns true if string A contains string B	sf.pproc.name=java and sf.pproc.cmdline contains org.apache.hadoop
A icontains B	Returns true if string A contains string B ignoring capitalization	sf.pproc.name=java and sf.pproc.cmdline icontains org.apache.hadooP
A pmatch B	Returns true if string A partial matches one of the elements in B. Note: B must be a list. Note: () can be used on B to merge multiple list objects into one list.	sf.proc.name pmatch (modify_passwd_binaries, verify_passwd_binaries, user_util_binaries)
exists A	Checks if A is not a zero value (i.e. 0 for int, "" for string)	exists sf.file.path

See the resources policies directory in github for examples. Feel free to contribute new and interesting rules through a github pull request.

User-defined Actions

User-defined actions are implemented via the golang plugin mechanism. Check the documentation on Action Plugins for a custom action plugin example.

3.5.6 Plugins

In addition to its core plugins, the processor also supports custom plugins that can be dynamically loaded into the processor via a compiled golang shared library using the golang plugin package. Custom plugins enable easy extension of the processor and the creation of custom pipelines tailored to specific use cases.

The processor supports four types of plugins:

- **drivers**: enable the ingestion of different telemetry sources into the processor pipeline.
- **processors**: enable the creation of custom data processing and analytic plugins to extend sf-processor pipelines.
- **handlers**: enable the creation of custom SysFlow record handling plugins.
- **actions**: enable the creation of custom action plugins to extend sf-processor's policy engine.

Pre-requisites

- Go 1.17 (if building locally, without the plugin builder)

Processor Plugins

User-defined plugins can be plugged and extend the sf-processor pipeline. These are the most generic type of plugins, from which all built-in processor plugins are build. Check the `core` package for examples. We have built-in processor plugins for flattening the telemetry stream, implementing a policy engine, and creating event exporters.

Interface

Processor plugins (or just plugins) are implemented via the golang plugin mechanism. A plugin must implement the following interface, defined in the `github.com/sysflow-telemetry/sf-apis/go/plugins` package.

```
// SFProcessor defines the SysFlow processor interface.
type SFProcessor interface {
  Register(pc SFPluginCache)
  Init(conf map[string]interface{}) error
  Process(ch interface{}, wg *sync.WaitGroup)
  GetName() string
  SetOutChan(ch []interface{})
  Cleanup()
}
```

The `Process` function is the main function of the plugin.It's where the "main loop" of the plugin should be implemented. It receives the input channel configured in the custom plugin's block in the pipeline configuration. It also received the pepeline thread WaitGroup. Custom processing code should be implemented using this function. `Init` is

called once, when the pipeline is loaded. `Cleanup` is called when the pipeline is terminated. `SetOutChannel` receives a slice with the output channels configured in the plugin's block in the pipeline configuration.

When loading a pipeline, sf-processor performs a series of health checks before the pipeline is enabled. If these health checks fail, the processor terminates. To enable health checks on custom plugins, implement the `Test` function defined in the interface below. For an example, check `core/exporter/exporter.go`.

```
// SFTestableProcessor defines a testable SysFlow processor interface.
type SFTestableProcessor interface {
  SFProcessor
  Test() (bool, error)
}
```

Example

A dynamic plugin example is provided in github. The core of the plugin is building an object that implements an SFProcessor interface. Such an implementation looks as follows:

```
package main

import (
    "sync"

    "github.com/sysflow-telemetry/sf-apis/go/logger"
    "github.com/sysflow-telemetry/sf-apis/go/plugins"
    "github.com/sysflow-telemetry/sf-apis/go/sfgo"
    "github.com/sysflow-telemetry/sf-processor/core/flattener"
)

const (
    pluginName string = "example"
)

// Plugin exports a symbol for this plugin.
var Plugin Example

// Example defines an example plugin.
type Example struct{}

// NewExample creates a new plugin instance.
func NewExample() plugins.SFProcessor {
    return new(Example)
}

// GetName returns the plugin name.
func (s *Example) GetName() string {
    return pluginName
}

// Init initializes the plugin with a configuration map.
func (s *Example) Init(conf map[string]interface{}) error {
    return nil
}
```

(continues on next page)

(continued from previous page)

```
// Register registers plugin to plugin cache.
func (s *Example) Register(pc plugins.SFPluginCache) {
    pc.AddProcessor(pluginName, NewExample)
}

// Process implements the main interface of the plugin.
func (s *Example) Process(ch interface{}, wg *sync.WaitGroup) {
    cha := ch.(*flattener.FlatChannel)
    record := cha.In
    logger.Trace.Println("Example channel capacity:", cap(record))
    defer wg.Done()
    logger.Trace.Println("Starting Example")
    for {
        fc, ok := <-record
        if !ok {
            logger.Trace.Println("Channel closed. Shutting down.")
            break
        }
        if fc.Ints[sfgo.SYSFLOW_IDX][sfgo.SF_REC_TYPE] == sfgo.PROC_EVT {
            logger.Info.Printf("Process Event: %s, %d", fc.Strs[sfgo.SYSFLOW_IDX][sfgo.
↪PROC_EXE_STR], fc.Ints[sfgo.SYSFLOW_IDX][sfgo.EV_PROC_TID_INT])
        }
    }
    logger.Trace.Println("Exiting Example")
}

// SetOutChan sets the output channel of the plugin.
func (s *Example) SetOutChan(ch []interface{}) {}

// Cleanup tears down plugin resources.
func (s *Example) Cleanup() {}

// This function is not run when module is used as a plugin.
func main() {}
```

The custom plugin must implement the following interface:

- `GetName()` - returns a lowercase string representing the plugin's label. This label is important, because it identifies the plugin in the `pipeline.json` file, enabling the processor to load the plugin. In the object above, this plugin is called `example`. Note that the label must be unique.
- `Init(conf map[string]interface{}) error` - used to initialize the plugin. The configuration map that is passed to the function stores all the configuration information defined in the plugin's definition inside `pipeline.json` (more on this later).
- `Register(pc plugins.SFPluginCache)` - this registers the plugin with the plugin cache of the processor.
  - `pc.AddProcessor(pluginName, <plugin constructor function>)` (required) - registers the plugin named `example` with the processor. You must define a constructor function using the convention `New<PluginName>` which is used to instantiate the plugin, and returns it as an `SFProcessor` interface - see `NewExample` for an example.
  - `pc.AddChannel(channelName, <output channel constructor function>)` (optional) - if your plugin is using a custom output channel of objects (i.e., the channel used to pass output objects from this

plugin to the next in the pipeline), it should be included in this plugin.

  * The `channelName` should be a lowercase unique label defining the channel type.
  * The constructor function should return a golang `interface{}` representing an object that as an `In` attribute of type `chan <ObjectToBePassed>`. We will call this object, a wrapped channel object going forward. For example, the channel object that passes sysflow objects is called SFChannel, and is defined here
  * For a complete example of defining an output channel, see `NewFlattenerChan` in the flattener as well as the `Register` function. The `FlatChannel` is defined here

- `Process(ch interface{}, wg *sync.WaitGroup)` - this function is launched by the processor as a go thread and is where the main plugin processing occurs. It takes a wrapped channel object, which acts as the input data source to the plugin (i.e., this is the channel that is configured as the input channel to the plugin in the pipeline.json). It also takes a sync.WaitGroup object, which is used to signal to the processor when the plugin has completed running (see `defer wg.Done()` in code). The processor must loop on the input channel, and do its analysis on each input record. In this case, the example plugin is reading flat records and printing them to the screen.
- `SetOutChan(ch []interface{})` - sets the wrapped channels that will serve as the output channels for the plugin. The output channels are instantiated by the processor, which is also in charge of stitching the plugins together. If the plugin is the last one in the chain, then this function can be left empty. See the `SetOutputChan` function in the flattener to see how an output channel is implemented.
- `Cleanup()` - Used to cleanup any resources. This function is called by the processor after the plugin `Process` function exits. One of the key items to close in the `Cleanup` function is the output channel using the golang `close()` function. Closing the output channel enables the pipeline to be torn down gracefully and in sequence.
- `main(){}` - this main method is not used by the plugin or processor. It's required by golang in order to be able to compile as a shared object.

To compile the example plugin, use the provided Makefile:

```
make -C plugins/processors/example
```

This will build the plugin and copy it into `resources/plugins/`.

To use the new plugin, use the configuration provided in github, which defines the following pipeline:

```
{
   "pipeline":[
     {
      "processor": "sysflowreader",
      "handler": "flattener",
      "in": "sysflow sysflowchan",
      "out": "flat flattenerchan"
     },
     {
      "processor": "example",
      "in": "flat flattenerchan"
     }
   ]
}
```

This pipeline contains two plugins:

- **The builtin `sysflowReader` plugin with flattener handler, which takes raw sysflow objects, and flattens them** into arrays of integers and strings for easier processing in certain plugins like the policy engine.

- The `example` plugin, which takes the flattened output from the sysflowreader plugin, and prints it the screen.

The key item to note is that the output channel (i.e., `out`) of `sysflowreader` matches the input channel (i.e., `in`) of the example plugin. This ensures that the plugins will be properly stitched together.

Build

The `example` plugin is a custom plugin that illustrates how to implement a minimal plugin that reads the records from the input channel and logs them to the standard output.

To run this example, in the root of sf-processor, build the processor and the example plugin. Note, this plugin's shared object is generated in `resources/plugins/example.so`.

```
make build && make -C plugins/processors/example
```

Then, run:

```
cd driver && ./sfprocessor -log=info -config=../plugins/processors/example/pipeline.
↪example.json ../resources/traces/tcp.sf
```

Plugin builder

To build the plugin for release, Go requires the code to be compiled with the exact package versions that the SysFlow processor was compiled with. The easiest way to achieve this is to use the pre-built `plugin-builder` Docker image in your build. This option also works for building plugins for deployment with the SysFlow binary packages.

Below is an example of how this can be achieved. Set $TAG to a SysFlow release (>=0.4.0), `edge`, or `dev`.

First, build the plugin:

```
docker run --rm \
    -v $(pwd)/plugins:/go/src/github.com/sysflow-telemetry/sf-processor/plugins \
    -v $(pwd)/resources:/go/src/github.com/sysflow-telemetry/sf-processor/resources \
    sysflowtelemetry/plugin-builder:$TAG \
    make -C /go/src/github.com/sysflow-telemetry/sf-processor/plugins/processors/example
```

To test it, run the pre-built processor with the example configuration and trace.

```
docker run --rm \
    -v $(pwd)/plugins:/usr/local/sysflow/plugins \
    -v $(pwd)/resources:/usr/local/sysflow/resources \
    -w /usr/local/sysflow/bin \
    --entrypoint=/usr/local/sysflow/bin/sfprocessor \
    sysflowtelemetry/sf-processor:$TAG \
    -log=info -config=../plugins/processors/example/pipeline.example.json ../resources/
↪traces/tcp.sf
```

The output on the above pre-recorded trace should look like this:

```
[Health] 2022/02/21 12:55:19 pipeline.go:246: Health checks: passed
[Info] 2022/02/21 12:55:19 main.go:147: Successfully loaded pipeline configuration
[Info] 2022/02/21 12:55:19 pipeline.go:170: Starting the processing pipeline
[Info] 2022/02/21 12:55:19 example.go:75: Process Event: ./server, 13823
[Info] 2022/02/21 12:55:19 example.go:75: Process Event: ./client, 13824
```

(continues on next page)

(continued from previous page)

```
[Info] 2022/02/21 12:55:19 example.go:75: Process Event: ./client, 13824
[Info] 2022/02/21 12:55:19 example.go:75: Process Event: ./server, 13823
```

Handler Plugins

User-defined handler modules can be plugged to the built-in SysFlow `processor` plugin to implement custom data processing and analytic pipelines.

Interface

Handlers are implemented via the golang plugin mechanism. A handler must implement the following interface, defined in the `github.com/sysflow-telemetry/sf-apis/go/plugins` package.

```
// SFHandler defines the SysFlow handler interface.
type SFHandler interface {
  RegisterChannel(pc SFPluginCache)
  RegisterHandler(hc SFHandlerCache)
  Init(conf map[string]interface{}) error
  IsEntityEnabled() bool
  HandleHeader(sf *CtxSysFlow, hdr *sfgo.SFHeader) error
  HandleContainer(sf *CtxSysFlow, cont *sfgo.Container) error
  HandleProcess(sf *CtxSysFlow, proc *sfgo.Process) error
  HandleFile(sf *CtxSysFlow, file *sfgo.File) error
  HandleNetFlow(sf *CtxSysFlow, nf *sfgo.NetworkFlow) error
  HandleNetEvt(sf *CtxSysFlow, ne *sfgo.NetworkEvent) error
  HandleFileFlow(sf *CtxSysFlow, ff *sfgo.FileFlow) error
  HandleFileEvt(sf *CtxSysFlow, fe *sfgo.FileEvent) error
  HandleProcFlow(sf *CtxSysFlow, pf *sfgo.ProcessFlow) error
  HandleProcEvt(sf *CtxSysFlow, pe *sfgo.ProcessEvent) error
  SetOutChan(ch []interface{})
  Cleanup()
}
```

Each `Handle*` function receives the current SysFlow record being processed along with its corresponding parsed record type. Custom processing code should be implemented using these functions.

Build

The `printer` handler is a pluggable handler that logs select SysFlow records to the standard output. This plugin doesn't define any output channels, so it acts as a plugin sink (last plugin in a pipeline).

To run this example, in the root of sf-processor, build the processor and the handler plugin. Note, this plugin's shared object is generated in `resources/handlers/printer.so`.

```
make build && make -C plugins/handlers/printer
```

Then, run:

```
cd driver && ./sfprocessor -log=info -config=../plugins/handlers/printer/pipeline.
↪printer.json ../resources/traces/tcp.sf
```

Plugin builder

To build the plugin for release, Go requires the code to be compiled with the exact package versions that the SysFlow processor was compiled with. The easiest way to achieve this is to use the pre-built `plugin-builder` Docker image in your build. This option also works for building plugins for deployment with the SysFlow binary packages.

Below is an example of how this can be achieved. Set $TAG to a SysFlow release (>=0.4.0), `edge`, or `dev`.

First, build the plugin:

```
docker run --rm \
    -v $(pwd)/plugins:/go/src/github.com/sysflow-telemetry/sf-processor/plugins \
    -v $(pwd)/resources:/go/src/github.com/sysflow-telemetry/sf-processor/resources \
    sysflowtelemetry/plugin-builder:$TAG \
    make -C /go/src/github.com/sysflow-telemetry/sf-processor/plugins/handlers/printer
```

To test it, run the pre-built processor with the example configuration and trace.

```
docker run --rm \
    -v $(pwd)/plugins:/usr/local/sysflow/plugins \
    -v $(pwd)/resources:/usr/local/sysflow/resources \
    -w /usr/local/sysflow/bin \
    --entrypoint=/usr/local/sysflow/bin/sfprocessor \
    sysflowtelemetry/sf-processor:$TAG \
    -log=info -config=../plugins/handlers/printer/pipeline.printer.json ../resources/
↪traces/tcp.sf
```

The output on the above pre-recorded trace should look like this:

```
[Info] 2022/02/21 15:39:58 printer.go:118: ProcEvt ./server, 13823
[Info] 2022/02/21 15:39:58 printer.go:100: FileFlow ./server, 3
[Info] 2022/02/21 15:39:58 printer.go:100: FileFlow ./server, 3
[Info] 2022/02/21 15:39:58 printer.go:118: ProcEvt ./client, 13824
[Info] 2022/02/21 15:39:58 printer.go:100: FileFlow ./client, 3
[Info] 2022/02/21 15:39:58 printer.go:100: FileFlow ./client, 3
[Info] 2022/02/21 15:39:58 printer.go:94: NetworkFlow ./client, 8080
[Info] 2022/02/21 15:39:58 printer.go:118: ProcEvt ./client, 13824
[Info] 2022/02/21 15:39:58 printer.go:94: NetworkFlow ./server, 8080
[Info] 2022/02/21 15:39:58 printer.go:118: ProcEvt ./server, 13823
```

Action Plugins

User-defined actions can be plugged to SysFlow's Policy Engine rule declarations to perform additional processing on matched records.

Interface

Actions are implemented via the golang plugin mechanism. An action must implement the following interface, defined in the `github.com/sysflow-telemetry/sf-processor/core/policyengine/engine` package.

```
// Prototype of an action function
type ActionFunc func(r *Record) error

// Action interface for user-defined actions
type Action interface {
        GetName() string
        GetFunc() ActionFunc
}
```

Actions have a name and an action function. Within a single policy engine instance, action names must be unique. User-defined actions cannot re-declare built-in actions. Reusing names of user-defined actions overwrites previously registered actions.

The action function receives the current record as an argument and thus has access to all record attributes. The action result can be stored in the record context via the context modifier methods.

Build

The `now` action is a pluggable action that creates a tag containing the current time in nanosecond precision.

First, in the root of sf-processor, build the processor and the action plugin. Note, this plugin's shared object is generated in `resources/actions/now.so`.

```
make build && make -C plugins/actions/example
```

Then, run:

```
cd driver && ./sfprocessor -log=quiet -config=../plugins/actions/example/pipeline.
↪actions.json ../resources/traces/tcp.sf
```

Plugin builder

To build the plugin for release, Go requires the code to be compiled with the exact package versions that the SysFlow processor was compiled with. The easiest way to achieve this is to use the pre-built `plugin-builder` Docker image in your build. This option also works for building plugins for deployment with the SysFlow binary packages.

Below is an example of how this can be achieved. Set $TAG to a SysFlow release (>=0.4.0), `edge`, or `dev`.

First, build the plugin:

```
docker run --rm \
    -v $(pwd)/plugins:/go/src/github.com/sysflow-telemetry/sf-processor/plugins \
```

(continues on next page)

(continued from previous page)

```
    -v $(pwd)/resources:/go/src/github.com/sysflow-telemetry/sf-processor/resources \
    sysflowtelemetry/plugin-builder:$TAG \
    make -C /go/src/github.com/sysflow-telemetry/sf-processor/plugins/actions/example
```

To test it, run the pre-built processor with the example configuration and trace.

```
docker run --rm \
    -v $(pwd)/plugins:/usr/local/sysflow/plugins \
    -v $(pwd)/resources:/usr/local/sysflow/resources \
    -w /usr/local/sysflow/bin \
    --entrypoint=/usr/local/sysflow/bin/sfprocessor \
    sysflowtelemetry/sf-processor:$TAG \
    -log=quiet -config=../plugins/actions/example/pipeline.actions.json ../resources/
↪traces/tcp.sf
```

In the output, observe that all records matching the policy speficied in `pipeline.actions.json` are tagged by action `now` with the tag `now_in_nanos`. For example:

```
{
  "version": 4,
  "endts": 0,
  "opflags": [
    "EXEC"
  ],
  ...
  "policies": [
    {
      "id": "Action example",
      "desc": "user-defined action example",
      "priority": 0
    }
  ],
  "tags": [
    "now_in_nanos:1645409122055957900"
  ]
}
```

3.5.7 Docker usage

Documentation and scripts for how to deploy the SysFlow Processor with docker compose can be found in here.

Processor environment

As mentioned in a previous section, all custom plugin attributes can be set using the following: `<PLUGIN NAME>_<CONFIG ATTRIBUTE NAME>` format. Note that the docker compose file sets several attributes including `EXPORTER_TYPE`, `EXPORTER_HOST` and `EXPORTER_PORT`.

The following are the default locations of the pipeline configuration and plugins directory:

- pipeline.json: `/usr/local/sysflow/conf/pipeline.json`
- drivers dir: `/usr/local/sysflow/resources/drivers`

- plugins dir: `/usr/local/sysflow/resources/plugins`
- handler dir: `/usr/local/sysflow/resources/handlers`
- actions dir: `/usr/local/sysflow/resources/actions`

The default configuration can be changed by setting up a virtual mounts mapping the host directories/files into the container using the volumes section of the sf-processor in the docker-compose.yaml.

```
sf-processor:
    container_name: sf-processor
    image: sysflowtelemetry/sf-processor:latest
    privileged: true
    volumes:
      ...
      - ./path/to/my.pipeline.json:/usr/local/sysflow/conf/pipeline.json
```

The policy location can be overwritten by setting the `POLICYENGINE_POLICIES` environment variable, which can point to a policy file or a directory containing policy files (must have yaml extension).

The docker container uses a default `filter.yaml` policy that outputs SysFlow records in json. You can use your own policy files from the host by mounting your policy directory into the container as follows, in which the custom pipeline points to the mounted policies.

```
sf-processor:
    container_name: sf-processor
    image: sysflowtelemetry/sf-processor:latest
    privileged: true
    volumes:
      ...
      - ./path/to/my.pipeline.json:/usr/local/sysflow/conf/pipeline.json
      - ./path/to/policies/:/usr/local/sysflow/resources/policies/
```

3.6 SysFlow Exporter (sf-exporter repo)

SysFlow exporter to export SysFlow traces to S3-compliant object stores.

> Note: For remote syslogging and other export formats and connectors, check the SysFlow processor project.

3.6.1 Build

This document describes how to build and run the application both inside a docker container and on a Linux host. Building and running the application inside a docker container is the easiest way to start. For convenience, skip the build step and pull pre-built images directly from Docker Hub.

To build the project, first clone the source code, with submodules:

```
git clone --recursive git@github.com:sysflow-telemetry/sf-exporter.git
```

To checkout submodules on an already cloned repo:

```
git submodule update --init --recursive
```

To build the docker image for the exporter locally, run:

```
docker build -t sf-exporter .
```

3.6.2 Docker usage

The easiest way to run the SysFlow exporter is from a Docker container, with host mount for the trace files to export. The following command shows how to run sf-exporter with trace files located in `/mnt/data` on the host.

```
docker run -d --rm --name sf-exporter \
    -e S3_ENDPOINT=<ip_address> \
    -e S3_BUCKET=<bucket_name> \
    -e S3_ACCESS_KEY=<access_key> \
    -e S3_SECRET_KEY=<secret_key> \
    -e NODE_IP=$HOSTNAME \
    -e INTERVAL=150 \
    -v /mnt/data:/mnt/data \
    sysflowtelemetry/sf-exporter
```

It's also possible to read S3's keys as docker secrets `s3_access_key` and `s3_secret_key`. Instructions for `docker compose` and `helm` deployments are available in here.

```
docker service create --name sf-exporter \
    -e NODE_IP=10.1.0.159 \
    -e INTERVAL=15 \
    --secret s3_access_key \
    --secret s3_secret_key \
    --mount type=bind,source=/mnt/data,destination=/mnt/data \
    sf-exporter:latest
```

The exporter is usually executed as a pod or docker-compose service together with the SysFlow collector. The exporter automatically removes exported files from the local filesystem it monitors. See the SysFlow deployments packages for more information.

3.6.3 Development

To build the exporter locally, run:

```
cd src & pip3 install -r requirements.txt
cd modules/sysflow/py3 & sudo python3 setup.py install
```

To run the exporter from the command line:

```
./exporter.py -h
usage: exporter.py [-h] [--exporttype {s3,local}] [--s3endpoint S3ENDPOINT]
                       [--s3port S3PORT] [--s3accesskey S3ACCESSKEY] [--s3secretkey␣
↪S3SECRETKEY]
                       [--s3bucket S3BUCKET] [--s3location S3LOCATION] [--s3prefix␣
↪S3PREFIX]
                       [--secure [SECURE]] [--scaninterval SCANINTERVAL] [--timeout␣
↪TIMEOUT]
                       [--agemin AGEMIN] [--log LOG] [--dir DIR] [--mode MODE] [--todir␣
↪TODIR]
```

(continues on next page)

(continued from previous page)

```
                        [--nodename NODENAME] [--nodeip NODEIP] [--podname PODNAME] [--
↪podip PODIP]
                        [--podservice PODSERVICE] [--podns PODNS] [--poduuid PODUUID] [--
↪clusterid CLUSTERID]

sf-exporter: watches and uploads monitoring files to object store.

optional arguments:
  -h, --help            show this help message and exit
  --exporttype {s3,local}
                        export type
  --s3endpoint S3ENDPOINT
                        s3 server address
  --s3port S3PORT       s3 server port
  --s3accesskey S3ACCESSKEY
                        s3 access key
  --s3secretkey S3SECRETKEY
                        s3 secret key
  --s3bucket S3BUCKET   target data bucket(s) comma delimited. number must match data␣
↪dirs
  --s3location S3LOCATION
                        target data bucket location
  --s3prefix S3PREFIX   s3 bucket prefix
  --secure [SECURE]     enables SSL connection
  --scaninterval SCANINTERVAL
                        interval between scans
  --timeout TIMEOUT     connection timeout
  --agemin AGEMIN       age in minutes to keep in case of repeated timeouts
  --log LOG             logging level for exporter: DEBUG, INFO, WARNING, ERROR, CRITICAL
  --dir DIR             data directory(s) comma delimited. number must match s3buckets
  --mode MODE           copy modes (move-del, cont-update, cont-update-recur) comma␣
↪delimited. number must match buckets, data dirs
  --todir TODIR         data directory
  --nodename NODENAME   exporter's node name
  --nodeip NODEIP       exporter's node IP
  --podname PODNAME     exporter's pod name
  --podip PODIP         exporter's pod IP
  --podservice PODSERVICE
                        exporter's pod service
  --podns PODNS         exporter's pod namespace
  --poduuid PODUUID     exporter's: pod UUID
  --clusterid CLUSTERID
                        exporter's: cluster ID
```

3.7 SysFlow APIs and Utilities (sf-apis repo)

3.7.1 SysFlow APIs and Utilities

SysFlow uses Apache Avro serialization to create compact records that can be processed by a wide variety of programming languages, and big data analytics platforms such as Apache Spark. Avro enables a user to generate programming stubs for serializing and deserializing data, using either Apache Avro IDL or Apache schema files.

Cloning source

The sf-apis project has been tested primarily on Ubuntu 16.04 and 18.04. The project will be tested on other flavors of UNIX in the future. This document describes how to build and run the application both on a linux host.

To build the project, first pull down the source code:

```
git clone git@github.com:sysflow-telemetry/sf-apis.git
```

Avro IDL and schema files

The Avro IDL files for SysFlow are available in the repository under `sf-apis/avro/avdl`, while the schema files are available under `sf-apis/avro/avsc`. The `avrogen` tool can be used to generate classes using the schema. See `sf-apis/avro/generateCClasses.sh` for an example of how to generate C++ headers from apache schema files.

SysFlow Avro C++

SysFlow C++ SysFlow objects and encoders/decoders are all available in `sf-apis/c++/sysflow/sysflow.hh`. `sf-collector/src/sysreader.cpp` provides a good example of how to read and process different SysFlow avro objects in C++. Note that one must install Apache Avro 1.9.1 cpp to run an application that includes `sysflow.hh`. The library file `-lavrocpp` must also be linked during compilation.

SysFlow Avro Python 3

SysFlow Python 3 APIs are generated with the avro-gen Python package. These classes are available in `sf-apis/py3`.

In order to install the SysFlow Python package:

```
cd sf-apis/py3
sudo python3 setup.py install
```

Please see the SysFlow Python API reference documents for more information on the modules and objects in the library.

SysFlow utilities

sysprint

`sysprint` is a tool written using the SysFlow Python API that will print out SysFlow traces from a file into several different formats including JSON, CSV, and tabular pretty print form. Not only will sysprint help you interact with SysFlow, it is also a good example for how to write new analytics tools using the SysFlow API.

```
usage: sysprint [-h] [-i {local,s3}] [-o {str,flatjson,json,csv}] [-w FILE]
                [-c FIELDS] [-f FILTER] [-l] [-e S3ENDPOINT] [-p S3PORT]
                [-a S3ACCESSKEY] [-s S3SECRETKEY] [-k] [-A]
                [--secure [SECURE]]
                path [path ...]

sysprint: a human-readable printer and format converter for Sysflow captures.

positional arguments:
  path                  list of paths or bucket names from where to read trace
                        files

optional arguments:
  -h, --help            show this help message and exit
  -i {local,s3}, --input {local,s3}
                        input type
  -o {str,flatjson,json,csv}, --output {str,flatjson,json,csv}
                        output format
  -w FILE, --file FILE  output file path
  -c FIELDS, --fields FIELDS
                        comma-separated list of sysflow fields to be printed
  -f FILTER, --filter FILTER
                        filter expression
  -l, --list            list available record attributes
  -e S3ENDPOINT, --s3endpoint S3ENDPOINT
                        s3 server address from where to read sysflows
  -p S3PORT, --s3port S3PORT
                        s3 server port
  -a S3ACCESSKEY, --s3accesskey S3ACCESSKEY
                        s3 access key
  -s S3SECRETKEY, --s3secretkey S3SECRETKEY
                        s3 secret key
  -k, --k8s             add pod related fields to output
  -A, --allfields       add all available fields to output
  --secure [SECURE]     indicates if SSL connection
```

3.7.2 SysFlow Python API Reference

SysFlow Reader API

class sysflow.reader.**FlattenedSFReader**(*filename*, *retEntities=False*)

FlattenedSFReader

This class loads a raw sysflow file, and links all Entities (header, process, container, files) with the current flow or event in the file. As a result, the user does not have to manage this information. This class supports the python iterator design pattern. Example Usage:

```
reader = FlattenedSFReader(trace)
head = 20 # max number of records to print
for i, (objtype, header, cont, pproc, proc, files, evt, flow) in enumerate(reader):
    exe = proc.exe
    pid = proc.oid.hpid if proc else ''
    evflow = evt or flow
    tid = evflow.tid if evflow else ''
    opFlags = utils.getOpFlagsStr(evflow.opFlags) if evflow else ''
    sTime = utils.getTimeStr(evflow.ts) if evflow else ''
    eTime = utils.getTimeStr(evflow.endTs) if flow else ''
    ret = evflow.ret if evt else ''
    res1 = ''
    if objtype == ObjectTypes.FILE_FLOW or objtype == ObjectTypes.FILE_EVT:
        res1 = files[0].path
    elif objtype == ObjectTypes.NET_FLOW:
        res1 = utils.getNetFlowStr(flow)
    numBReads = evflow.numRRecvBytes if flow else ''
    numBWrites = evflow.numWSendBytes if flow else ''
    res2 = files[1].path if files and files[1] else ''
    cont = cont.id if cont else ''
    print("|{0:30}|{1:9}|{2:26}|{3:26}|{4:30}|{5:8}|{6:8}|".format(exe, opFlags,␣
↪sTime, eTime, res1, numBReads, numBWrites))
    if i == head:
        break
```

Parameters

- **filename** (*str*) – the name of the sysflow file to be read.
- **retEntities** (*bool*) – If True, the reader will return entity objects by themselves as they are seen in the sysflow file. In this case, all other objects will be set to None

Iterator

Reader returns a tuple of objects in the following order:

objtype (*sysflow.objtypes.ObjectTypes*) The type of entity or flow returned.

header (sysflow.entity.SFHeader) The header entity of the file.

pod (sysflow.entity.Pod) The pod associated with the flow/evt, or None if no pod.

cont (sysflow.entity.Container) The container associated with the flow/evt, or None if no container.

pproc (sysflow.entity.Process) The parent process associated with the flow/evt.

proc (sysflow.entity.Process) The process associated with the flow/evt.

files (tuple of `sysflow.entity.File`) Any files associated with the flow/evt.

evt (`sysflow.event.{ProcessEvent,FileEvent}`) If the record is an event, it will be returned here. Otherwise this variable will be None. objtype will indicate the type of event.

flow (`sysflow.flow.{NetworkFlow,FileFlow}`) If the record is a flow, it will be returned here. Otherwise this variable will be None. objtype will indicate the type of flow.

`getProcess`(*oid*)

Returns a Process Object given a process object id.

Parameters
oid (*sysflow.type.OID*) – the object id of the Process Object requested

Return type
sysflow.entity.Process

Returns
the desired process object or None if no process object is available.

`class` sysflow.reader.`NestedNamespace`(***kwargs*)

`class` sysflow.reader.`SFReader`(*filename*)

SFReader

This class loads a raw sysflow file, and returns each entity/flow one by one. It is the user's responsibility to link the related objects together through the OID. This class supports the python iterator design pattern. Example Usage:

```
reader = SFReader("./sysflowfile.sf")
for name, sf in reader:
    if name == "sysflow.entity.SFHeader":
       //do something with the header object
    elif name == "sysflow.entity.Container":
       //do something with the container object
    elif name == "sysflow.entity.Process":
       //do something with the Process object
    ....
```

Parameters
filename (*str*) – the name of the sysflow file to be read.

SysFlow Formatter API

`class` sysflow.formatter.`SFFormatter`(*reader*, *defs=[]*)

SFFormatter

This class takes a *FlattenedSFReader*, and exports SysFlow as either JSON, CSV or Pretty Print . Example Usage:

```
reader = FlattenedSFReader(trace, False)
formatter = SFFormatter(reader)
fields=args.fields.split(',') if args.fields else None
if args.output == 'json':
    if args.file is not None:
        formatter.toJsonFile(args.file, fields=fields)
```

(continues on next page)

(continued from previous page)

```
    else:
        formatter.toJsonStdOut(fields=fields)
elif args.output == 'csv' and args.file is not None:
    formatter.toCsvFile(args.file, fields=fields)
elif args.output == 'str':
    formatter.toStdOut(fields=fields)
```

Parameters

- **reader** (sysflow.reader.FlattenedSFReader) – A reader representing the sysflow file being read.
- **defs** (*list*) – A list of paths to filter definitions.

applyFuncJson(*func*, *fields=None*, *expr=None*)

Enables a delegate function to be applied to each JSON record read.

Parameters

- **func** (*function*) – delegate function of the form func(str)
- **fields** (*list*) – a list of the SysFlow fields to be exported in JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

enableAllFields()

Enables all available fields to be added to the output by default.

enableK8sEventFields()

Enables fields related to k8s events be added to the output by default.

enablePodFields()

Enables fields related to pods to be added to the output by default.

getFields()

Returns a list with available SysFlow fields and their descriptions.

toCsvFile(*path*, *fields=None*, *header=True*, *expr=None*)

Writes SysFlow to CSV file.

Parameters

- **path** (*str*) – the full path of the output file.
- **fields** (*list*) – a list of the SysFlow fields to be exported in the JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

toDataframe(*fields=None*, *expr=None*)

Enables a delegate function to be applied to each JSON record read.

Parameters

- **func** (*function*) – delegate function of the form func(str)
- **fields** (*list*) – a list of the SysFlow fields to be exported in the JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

toJson(*fields=None*, *flat=False*, *expr=None*)

Writes SysFlow as JSON object.

Parameters

- **fields** (*list*) – a list of the SysFlow fields to be exported in JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

Flat
specifies if JSON output should be flattened

toJsonFile(*path*, *fields=None*, *flat=False*, *expr=None*)

Writes SysFlow to JSON file.

Parameters

- **path** (*str*) – the full path of the output file.
- **fields** (*list*) – a list of the SysFlow fields to be exported in JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

Flat
specifies if JSON output should be flattened

toJsonStdOut(*fields=None*, *flat=False*, *expr=None*)

Writes SysFlow as JSON to stdout.

Parameters

- **fields** (*list*) – a list of the SysFlow fields to be exported in JSON. See formatter.py for a list of fields
- **expr** (*str*) – a sfql filter expression

Flat
specifies if JSON output should be flattened

toStdOut(*fields=['ts_uts', 'type', 'proc.exe', 'proc.args', 'pproc.pid', 'proc.pid', 'proc.tid', 'opflags', 'res', 'flow.rbytes', 'flow.wbytes', 'container.id']*, *pretty_headers=True*, *showindex=True*, *expr=None*)

Writes SysFlow as a tabular pretty print form to stdout.

Parameters

- **fields** (*list*) – a list of the SysFlow fields to be exported in the JSON. See formatter.py for a list of fields
- **pretty_headers** (*bool*) – print table headers in pretty format.
- **showindex** (*bool*) – show record number.
- **expr** (*str*) – a sfql filter expression

SysFlow Object Types

class sysflow.objtypes.**ObjectTypes**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

ObjectTypes

Enumeration representing each of the object types:
HEADER = 0, CONT = 1, PROC = 2, FILE = 3, PROC_EVT = 4, NET_FLOW = 5, FILE_FLOW = 6, FILE_EVT = 7 PROC_FLOW = 8 POD = 9 K8S_EVT = 10

SysFlow Utils API

sysflow.utils.**getEnvStr**(*env*)

Converts an array of environment variables into a string representation.

Parameters
env (*str[]*) – An array of environment variables.

Return type
str

Returns
A concatenated string representation of the environment variables array.

sysflow.utils.**getIpIntStr**(*ipInt*)

Converts an IP address in host order integer to a string representation.

Parameters
ipInt – an IP address integer

Return type
str

Returns
A string representation of the IP address

sysflow.utils.**getNetFlowStr**(*nf*)

Converts a NetworkFlow into a string representation.

Parameters
nf (*sysflow.schema_classes.SchemaClasses.sysflow.flow.NetworkFlowClass*) – a NetworkFlow object.

Return type
str

Returns
A string representation of the NetworkFlow in form (sip:sport-dip:dport).

sysflow.utils.**getOpFlags**(*opFlags*)

Converts a sysflow operations flag bitmap into a set representation.

Parameters
opflag (*int*) – An operations bitmap from a flow or event.

Return type
set

Returns
A set representation of the operations bitmap.

sysflow.utils.**getOpFlagsStr**(*opFlags*)

Converts a sysflow operations flag bitmap into a string representation.

Parameters
opflag (*int*) – An operations bitmap from a flow or event.

Return type
str

Returns
A string representation of the operations bitmap.

sysflow.utils.**getOpStr**(*opFlags*)

Converts a sysflow operations into a string representation.

Parameters
opflag (*int*) – An operations bitmap from a flow or event.

Return type
str

Returns
A string representation of the operations bitmap.

sysflow.utils.**getOpenFlags**(*openFlags*)

Converts a sysflow open modes flag bitmap into a set representation.

Parameters
opflag – An open modes bitmap from a flow or event.

Return type
set

Returns
A set representation of the open modes bitmap.

sysflow.utils.**getTimeStr**(*ts*)

Converts a nanosecond ts into a string representation.

Parameters
ts (*int*) – A nanosecond epoch.

Return type
str

Returns
A string representation of the timestamp in %m/%d/%YT%H:%M:%S.%f format.

sysflow.utils.**getTimeStrIso8601**(*ts*)

Converts a nanosecond ts into a string representation in UTC time zone.

Parameters
ts (*int*) – A nanosecond epoch.

Return type
str

Returns
A string representation of the timestamp in ISO 8601 format.

SysFlow Graphlet API

class sysflow.graphlet.**Edge**(*n1*, *n2*, *label*)

Edge

This class represents a graph edge, and acts as a super class for specific edges.

Parameters
edge (*sysflow.Edge*) – an abstract edge object.

class sysflow.graphlet.**EvtEdge**(*n1*, *n2*, *label*)

EvtEdge

This class represents a graph event edge. It is used for sysflow event objects and subclasses Edge.

Parameters
evtedge (*sysflow.EvtEdge*) – an edge object representing a sysflow evt.

class sysflow.graphlet.**FileFlowNode**(*oid*, *exe*, *args*)

FileFlowNode

This class represents a fileflow node.

Parameters
ff (*sysflow.FileFlow*) – a fileflow node object.

class sysflow.graphlet.**FlowEdge**(*n1*, *n2*, *label*)

FlowEdge

This class represents a graph flow edge. It is used for sysflow flow objects and subclasses Edge.

Parameters
flowedge (*sysflow.FlowEdge*) – an edge object representing a sysflow flow.

class sysflow.graphlet.**Graphlet**(*path*, *expr=None*, *defs=[]*)

Graphlet

This class takes a path pointing to a sysflow trace or a directory containing sysflow traces.

Example Usage:

```
# basic usage
g1 = Graphlet('data/')
g1.view()

# filtering and enrichment with policies
ioc1 = 'proc.exe = /usr/bin/scp'
g1 = Graphlet('data/', ioc1, ['policies/ttps.yaml'])
g1.view()
```

Parameters
graphlet (*sysflow.Graphlet*) – A compact provenance graph representation based on sysflow traces.

associatedMitigations(*oid=None*)

Returns a dataframe containing the set of MITRE mitigations associated with TTPs annotated in the graph.

Parameters
oid (*object ID string*) – a node ID filter.

compare(*other*, *withoid=False*, *peek=True*, *peeksize=3*, *flows=True*, *ttps=False*)

Compares the graph to another graph (using a simple graph difference), returning a graph slice.

Parameters

- **withoid** (*boolean*) – indicates whether to show the node ID.
- **peek** (*boolean*) – indicates whether to show details about the records associated with the nodes.
- **peeksize** (*integer*) – the number of node records to show.
- **flows** (*boolean*) – indicates whether to show flow nodes.
- **ttps** (*boolean*) – indicates whether to show tags.

countermeasures(*oid=None*)

Returns a dataframe containing the set of MITRE d3fend defenses associated with TTPs annotated in the graph.

Parameters

oid (*object ID string*) – a node ID filter.

data(*oid=None*)

Returns a dataframe containing the underlying data (sysflow records) of the graph.

Parameters

oid (*object ID string*) – a node ID filter.

df(*oid=None*)

Returns a dataframe containing a summary of the graph node IDs and process metadata associated with them.

Parameters

oid (*object ID string*) – a node ID filter.

mitigations(*oid=None*, *details=False*)

Returns a dataframe containing the summary set of MITRE mitigations associated with TTPs annotated in the graph.

Parameters

oid (*object ID string*) – a node ID filter.

tags(*oid=None*)

Returns a dataframe containing the set of (enrichment) tags in the graph.

Parameters

oid (*object ID string*) – a node ID filter.

ttps(*oid=None*, *details=False*)

Returns a dataframe containing the set of MITRE TTP tags in the graph (e.g., as enriched by the ttps.yaml policy provided with the SysFlow processor).

Parameters

- **oid** (*object ID string*) – a node ID filter.
- **details** (*boolean*) – indicates whether to include complete TTP metadata in the dataframe.

view(*withoid=False*, *peek=True*, *peeksize=3*, *flows=True*, *ttps=False*)

Visualizes the graph in dot format.

Parameters

- **withoid** (*boolean*) – indicates whether to show the node ID.
- **peek** (*boolean*) – indicates whether to show details about the records associated with the nodes.
- **peeksize** (*integer*) – the number of underlying sysflow records to show in the node.
- **flows** (*boolean*) – indicates whether to show flow nodes.
- **ttps** (*boolean*) – indicates whether to show tags.

class sysflow.graphlet.**NetFlowNode**(*oid*, *exe*, *args*)

NetFlowNode

This class represents a netflow node.

Parameters

nf (*sysflow.NetFlow*) – a netflow node object.

class sysflow.graphlet.**Node**(*oid*)

Node

This class represents a graph node, and acts as a super class for specific nodes.

Parameters

node (*sysflow.Node*) – an abstract node object.

class sysflow.graphlet.**ProcessNode**(*oid*, *exe*, *args*, *uid*, *user*, *gid*, *group*, *tty*)

ProcessNode

This class represents a process node.

Parameters

proc (*sysflow.ProcessNode*) – a process node object.

SysFlow QL API

class sysflow.sfql.**SfqlInterpreter**(*query: str = None*, *paths: list = []*, *inputs: list = []*)

SfqlInterpreter

This class takes a sfql expression (and optionally a file containining a library of lists and macros) and produces a predicate expression that can be matched against sysflow records. Example Usage:

```
# using 'filter' to filter the input stream
reader = FlattenedSFReader('trace.sf')
interpreter = SfqlInterpreter()
query = '- sfql: type = FF'
for r in interpreter.filter(reader, query):
    print(r)
```

Parameters

interpreter (*sysflow.SfqlInterpreter*) – An interpreter for executing sfql expressions.

compile(*query: str = None*, *paths: list = []*, *inputs: list = []*)

Compile sfql into a predicate expression to match sysflow records.

Parameters

- **query** (*str*) – sfql query.
- **paths** (*list*) – a list of paths to file containing sfql list and macro definitions.
- **inputs** (*list*) – a list of input streams from where to read sfql list and macro definitions.

enrich(*t: T*)

Process flattened sysflow record t based on policies.

evaluate(*t: T*, *query: str = None*, *paths: list = []*) → bool

Evaluate sfql expression against flattened sysflow record t.

Parameters

- **reader** – individual sysflow record
- **query** (*str*) – sfql query.
- **paths** (*list*) – a list of paths to file containing sfql list and macro definitions.

filter(*reader*, *query: str = None*, *paths: list = []*)

Filter iterable reader according to sfql expression.

Parameters

- **reader** (FlattenedSFReader) – sysflow reader
- **query** (*str*) – sfql query.
- **paths** (*list*) – a list of paths to file containing sfql list and macro definitions.

getAttributes()

Return list of attributes supported by sfql.

class sysflow.sfql.**SfqlMapper**

3.8 Deployments (sf-deployments repo)

SysFlow can be deployed using Docker Compose, Helm, and binary packages.

3.8.1 Docker Compose

This repository contains utility scripts to deploy a docker telemetry stack.

Pre-requisites

- Docker (installing Docker)
- Docker Compose (installing Compose)

To guarantee a smooth deployment, the kernel headers must be installed in the host operating system.

This can usually be done on Debian-like distributions with:

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

Or, on RHEL-like distributions:

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

Deploy SysFlow

Three deployment configurations are described below: *local* (collector-only), *batch* export mode, and *stream* export mode. The local deployment stores collected traces on the local filesystem and the full stack deployments export the collected traces to a S3-compatible object storage server or streams SysFlow records to remote syslog server or ELK (additional exporters can be implemented as plugins).

Setup

Clone this repository and change directory as follows:

```
git clone https://github.com/sysflow-telemetry/sf-deployments.git
cd sf-deployments/docker
```

Local collection probe only

This deployment will install the Sysflow collection probe only, i.e., without an exporter to an external data store (e.g., S3). See below for the deploytment of the full telemetry stack.

To start the telemetry probe (collector only):

```
docker-compose -f docker-compose.collector.yml up
```

> Tip: add container.type!=host to `FILTER` string located in `./config/.env.collector` to filter out host (non-containerized) events.

To stop the collection probe:

```
docker-compose -f docker-compose.collector.yml down
```

Batch export

This deployment configuration includes the SysFlow Collector and S3 Exporter.

First, create the docker secrets used to connect to the S3 object store:

```
echo "<s3 access key>" > ./secrets/access_key
echo "<s3 secret key>" > ./secrets/secret_key
```

Then, configure the S3 endpoint in the exporter settings (default values point to a local minio object store described below). Exporter configuration is located in `./config/.env.exporter`. Collector settings can be changed in `./config/.env.collector`. Additional settings can be configured directly in compose file.

To start the telemetry stack:

```
docker-compose -f docker-compose.exporter.yml up
```

To stop the telemetry stack:

```
docker-compose -f docker-compose.exporter.yml down
```

To start the telemetry stack with a local minio object store:

```
docker-compose -f docker-compose.minio.yml -f docker-compose.exporter.yml up
```

To stop the local minio instance and the telemetry stack:

```
docker-compose -f docker-compose.minio.yml -f docker-compose.exporter.yml down
```

Stream processing

This deployment configuration includes the SysFlow Collector and Processor with rsyslog exporter. Alternatively, you can change the Processor configuration to stream events to ELK, or any other custom exporter plugin. Check the Processor's exporter configuration for details on how to configure the exporter to stream events to other backends.

First, configure the rsyslog endpoint in the processor settings. Processor configuration is located in `./config/.env.processor`. Collector settings can be changed in `./config/.env.collector`. Additional settings can be configured directly in compose file.

To start the telemetry stack:

```
docker-compose -f docker-compose.processor.yml up
```

To stop the telemetry stack:

```
docker-compose -f docker-compose.processor.yml down
```

Sysflow trace inspection

Run `sysprint` and point it to a trace file. In the examples below, `sysprint` is an alias for:

```
docker run --rm -v /mnt/data:/mnt/data sysflowtelemetry/sysprint
```

Tabular output

```
sysprint /mnt/data/<trace name>
```

JSON output

```
sysprint -o json /mnt/data/<trace name>
```

CSV output

```
sysprint -o csv /mnt/data/<trace name>
```

Inspect traces exported to an object store

```
sysprint -i s3 -c <s3_endpoint> -a <s3_access_key> -s <s3_secret_key> <bucket_name>
```

Tip: see all options of the `sysprint` utility with `-h` option.

Inspect example traces

Sample trace files are provided in `sf-collector/tests`. Copy them into `/mnt/data` to inspect inside sysprint's environment.

```
sysprint /mnt/data/tests/client-server/tcp-client-server.sf
```

Tip: other samples can be found in the tests directory

Analyzing collected traces

A Jupyter environment is also available for inspecting and implementing analytic notebooks on collected SysFlow data. It includes APIs for data manipulation using Pandas dataframes and a native query language (`sfql`) with macro support. To start it locally with example notebooks, run:

```
git clone https://github.com/sysflow-telemetry/sf-apis.git && cd sf-apis
docker run --rm -d --name sfnb -v $(pwd)/pynb:/home/jovyan/work -p 8888:8888␣
↪sysflowtelemetry/sfnb
```

Then, open a web browser and point it to `http://localhost:8888` (alternatively, the remote server name or IP where the notebook is hosted). To obtain the notebook authentication token, run `docker logs sfnb`.

3.8.2 Helm Charts

Helm charts are provided to facilitate the deployment and configuration of SysFlow on Kubernetes.

These charts have been tested on minikube and IBM Cloud Kubernetes Service. They shoud work on vanilla Kubernetes installations but it's possible that minor differences in how authentication is handled by different cloud providers require small modifications to the charts.

> These scripts have been tested with helm versions 2 and 3. Some helm commands may not work with other versions of helm.

Prerequisites

- kubectl (installing kubectl)
- Helm (installing helm)
- Docker (optional)

Install minikube (optional)

To deploy SysFlow on a local Kubernetes instance (for development or testing), start by installing minikube in your macOS, Linux, or Windows system.

For example, to install minikube in Linux distributions, run:

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

Then, start your cluster:

```
minikube start
```

> Note: to install SysFlow on minikube, set `sfcollector.ebpf` and `sfcollector.mountEtc` to `true` in `values.yaml` located inside each chart.

Check the minikube docs for additional installation options.

> Tip: run `eval $(minikube docker-env)` to allow your Docker CLI to connect to minikube's Docker environment.

The recommended driver for minikube is VirtualBox. Check the VirtualBox docs for installation instructions for your environment.

> A note about Docker pull limits: If you run into an error when deploying SysFlow on minikube, check the logs to see if it's related to the Docker pull limit being reached. It most likely is. To work around this inconvenience, connect to Minikube's Docker environment (see above), log into Docker with `docker login` command, and pull the desired images manually, before installing the helm charts. Make sure the images pull policies are set to the default value `IfNotPresent`.

Deploy SysFlow

The SysFlow agent can be deployed in S3 (batch) or rsyslog (stream) export configurations.

Setup

Clone this repository and change directory as follows:

```
git clone https://github.com/sysflow-telemetry/sf-deployments.git
cd sf-deployments/helm
```

Installing the SysFlow agent with S3 Exporter

In this configuration, SysFlow exports the collected telemetry as trace files (batches of SysFlow records) to any S3-compliant object storage service.

This chart is located in `charts/sf-exporter-chart`, which deploys the SysFlow Collector and Exporter as a daemonset. The collector monitors the node, and writes trace files to a shared memory volume `/mnt/data` which the exporter manages and reads from to push completed traces to a S3-compliant object storage. The `/mnt/data/` is mapped to a tmpfs filesystem, and you can specify its size using the `tmpfsSize`.

Installation scripts are provided to make installation easier. These scripts set up the environment including k8s secrets for S3 authentication. To connect to an S3-compliant data store, first take note of which port the S3 data store (`s3Port`) is configured. Minio installations listen on port 9000 by default. Also, if TLS is enabled on the S3 datastore, ensure `s3Secure` is `true`. Ensure that the `s3Bucket` is set to the desired S3 bucket location. The `s3Location` (aka `s3_region`), `s3AccessKey` and `s3SecretKey` and `s3Endpoint` are each passed in through the installation script if you use it.

To deploy the SysFlow agent with S3 export:

```
./scripts/installExporterChart.sh <s3_region> <s3_access_key> <s3_secret_key> <s3_
↪endpoint> <s3_bucket>
```

Installing the SysFlow agent with rsyslog exporter

In this configuration, SysFlow exports the collected telemetry as events streamed to a rsyslog collector. This deployment enables the creation of customized edge pipelines, and offers a built-in policy engine to filter, enrich, and alert on SysFlow records.

This chart is located in `charts/sf-processor-chart`, which deploys the SysFlow Collector and Processor as a daemonset. The collector monitors the node, and streams SysFlow records to the processor, which executes a configurable edge analytic pipeline and export events to a rsyslog endpoint.

To deploy the SysFlow agent with rsyslog export:

```
./scripts/installProcessorChart.sh <syslog_host> <syslog_port> <syslog_proto>
```

Checking installation

To check that the install worked, run:

```
kubectl get pods -n sysflow
```

To check the log output of the collector container in a pod:

```
kubectl logs -f -c sfcollector <podname>  -n sysflow
```

To check the log output of the exporter container in a pod:

```
kubectl logs -f -c sfexporter <podname>  -n sysflow
```

To check the log output of the processor container in a pod:

```
kubectl logs -f -c sfprocessor <podname>  -n sysflow
```

Removing the SysFlow agent

To remove the SysFlow agent:

```
./scripts/deleteChart.sh
```

Advanced customizations

Most of the defaults should work out of the box. The collector is currently set to rotating files in 5 min intervals (or 300 seconds). CGroup resource limits can be set on the collector, exporter, and processor to limit resource usage. These can be adjusted depending on requirements and resources limitations.

Note: `sfcollector.dropMode` is set to `true` by default for performance considerations.

Kubernetes can use different container runtimes. Older versions used the docker runtime; however, newer versions typically run either containerd or crio. It's important to know which runtime you have if you want to get the full benefits of SysFlow. You tell the collector which runtime you are using based on the sock file you refer to in the `criPath` variable. If you are using the `docker` runtime, leave `criPath` blank. If you are using containerd, set `criPath` to "/var/run/containerd/containerd.sock" and if you are using crio, set `criPath` to "/var/run/crio/crio.sock". If SysFlow files are empty or the container name variable is set to `incomplete` in SysFlow traces, this typically means that the runtime socket is not connected properly.

Note: the installation script installs the pods into a K8s namespace called `sysflow`.

Below is the list of customizable attributes for the charts, organized by component. These can be modified directly into the `values.yaml` located in each chart's directory. They can also be set directly into the helm command invoked by our installation scripts through `--set <attribute>=<value>` parameters.

SysFlow Collector

parameter	description	default
sfcollector.imagep	Pull policy for image (Always|Never|IfNotPresent)	Always
sfcollector.reposit	Image repository	sysflowtelemetry/sf-collector
sfcollector.tag	Image tag	latest
sfcollector.interva	Interval in seconds to roll new trace files	300
sfcollector.outDir	Directory in which collector writes trace files	/mnt/data/
sfcollector.filter	Filter expression	“"container.type!=host and container.name!=sfexporter and container.name!=sfcollector"”
sfcollector.criPath	Container runtime socket path. Use this “/var/run/containerd/containerd.sock”if running containerd runtime. Use “/var/run/crio/crio.sock” if running crio runtime.	“”
sfcollector.dropM	Drop mode filters syscalls in the kernel before they are passed up to the collector, resulting in much better performance and fewer event drops. Note: It filters mmap system calls from the event stream.	true
sfcollector.fileOnl	Filters out any descriptor that is not a file, including unix sockets and pipes	false
sfcollector.procFl	Enables the creation of process flows	false
sfcollector.readM	Sets mode for reads: `0` enables recording all file reads as flows. `1` disables all file reads. `2` disables recording file reads to noisy directories: “/proc/”, “/dev/”, “/sys/”, “//sys/”, “/lib/”, “/lib64/”, “/usr/lib/”, “/usr/lib64/”.	0
sfcollector.ebpf	Enables ebpf probe (required for minikube deployment)	false
sfcollector.mountE	Mounts etc directory in container (required for minikube and Google COS)	false
sfcollector.collecti	Template modes for enabling certain system calls. Currently supports 3 modes: flow” - full sysflows, “consume” - file reads, writes, closes turned off, “nofiles” - no fileevents or fileflows	flow
sfcollector.enable	When enabled, logs stats on containers, processes, networkflows, fileflows and records written at interval set by “interval” attribute	false

SysFlow Exporter

parameter	description	default
sfexporter.ena	Indicates whether the exporter will be used in the k8s deployment	false
sfexporter.im	Pull policy for image (Always|Never|IfNotPresent)	Always
sfexporter.rep	Image repository	sysflowtelemetry/sf-exporter
sfexporter.tag	Image tag	latest
sfexporter.log	Exporter logging level. Can be DEBUG, INFO, WARNING, ERROR, CRITICAL	INFO
sfexporter.typ	Type of trace export - “s3” to export to S3 storage, “local” for local copy	s3
sfexporter.int	Interval in seconds to check whether to export trace files	5
sfexporter.ou	Directory shared between the collector and exporter and where collector writes	/mnt/data/
sfexporter.dir	Directories (comma separated) from which exporter will copy	/mnt/data
sfexporter.toI	Directories (comma separated) to copy trace too - only used when type = “local”. Must have same number of entries as dirs attribute	commented out
sfexporter.mo	modes of copy (comma separated) move-del - move and delete file once finished writing - this is the only mode local copy supports. cont-update - continuously copy file over at interval (s3), cont-update-recur - continously update a directory structure recursively (s3). Must have same number of entries as dirs attribute	move-del
sfexporter.s3I	S3 host address (only used when type s3)	“<ip address>”
sfexporter.s3I	S3 port (only used when type s3)	443
sfexporter.s3I	S3 bucket where to push traces (only used when type s3). Can be a comma separated list of buckets. Must have same number of entries as dirs attribute	“<s3 bucket>”
sfexporter.s3I	S3 location (only used when type s3)	“<s3 region>”
sfexporter.s3A	S3 access key (only used when type s3)	“<s3 access key>”
sfexporter.s3S	S3 secret key (only used when type s3)	“<s3 secret key>”
sfexporter.s3S	S3 connection, `true` if TLS-enabled, `false` otherwise (only used when type s3)	false

SysFlow Processor

parameter	description	default
sfprocessor.imagepullpolicy	Pull policy for image (Always|Never|IfNotPresent)	Always
sfprocessor.repository	Image repository	sysflowtelemetry/sfprocessor
sfprocessor.tag	Image tag	latest
sfprocessor.export	Export type (`terminal`|`file`|`syslog`)	syslog
sfprocessor.override	Override processor exporter in pipeline.json with values.yaml settings	true
sfprocessor.syslogHost	rsyslog host address	localhost
sfprocessor.syslogPort	rsyslog port	514
sfprocessor.syslogProto	rsyslog protocol (`udp`|`tcp`|`tcp+tls`)	tcp
sfprocessor.configMapEnable	'true' if using config map for policy configs	'true'
sfprocessor.findingsDir	Directory to which raw findings are written. Must be the same as the findings.path value in the pipeline.json	/mnt/findings

3.8.3 Binary packages (deb|rpm)

SysFlow can be deployed directly on the host using its binary packages (since SysFlow 0.4.0).

We package SysFlow for debian- and rpm-based distros.

Debian distributions

Download the SysFlow packages (set `$VERSION` to a Sysflow release >=0.4.1):

```
wget https://github.com/sysflow-telemetry/sf-collector/releases/download/$VERSION/
↪sfcollector-$VERSION-x86_64.deb \
    https://github.com/sysflow-telemetry/sf-processor/releases/download/$VERSION/
↪sfprocessor-$VERSION-x86_64.deb
```

Install pre-requisites:

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

Install the SysFlow packages:

```
dpkg -i sfcollector-$VERSION-x86_64.deb sfprocessor-$VERSION-x86_64.deb
```

RPM distributions

Download the SysFlow packages (set `$VERSION` to a Sysflow release >=0.4.1):

```
wget https://github.com/sysflow-telemetry/sf-collector/releases/download/$VERSION/
↪sfcollector-$VERSION-x86_64.rpm \
    https://github.com/sysflow-telemetry/sf-processor/releases/download/$VERSION/
↪sfprocessor-$VERSION-x86_64.rpm
```

Install pre-requisites (Instructions for Rhel8 below):

```
subscription-manager repos --enable="codeready-builder-for-rhel-8-$(/bin/arch)-rpms"
dnf -y update
dnf -y install \
    kernel-devel-$(uname -r) \
    llvm-toolset
```

Install the SysFlow packages:

```
rmp -i sfcollector-$VERSION-x86_64.rpm sfprocessor-$VERSION-x86_64.rpm
```

Running

Start the SysFlow systemd service:

```
sysflow start
```

Check SysFlow service status:

```
sysflow status
```

Stop the SysFlow service:

```
sysflow stop
```

Configuration

Configuration options can be changed in `/etc/sysflow`. The Processor configuration is located in `/etc/sysflow/pipelines/pipeline.local.json` and can be used to change the processor configuration from its default settings. The Collector and systemd service configurations are located in `/etc/sysflow/conf/sysflow.env`.

3.9 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.
```

(continues on next page)

(continued from previous page)

```
   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
```

(continues on next page)

(continued from previous page)

```
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
```

(continues on next page)

(continued from previous page)

```
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.
```

(continues on next page)

(continued from previous page)

```
9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

3.10 Contributing

3.10.1 Contributing In General

Our project welcomes external contributions.

To contribute code or documentation, please submit a pull request to the proper github repositories.

A good way to familiarize yourself with the codebase and contribution process is to look for and tackle low-hanging fruit in the github issue trackers associated with projects. Before embarking on a more ambitious contribution, please quickly *get in touch* with us.

Note: We appreciate your effort, and want to avoid a situation where a contribution requires extensive rework (by you or by us), sits in backlog for a long time, or cannot be accepted at all!

Proposing new features

If you would like to implement a new feature, please raise an issue in the appropriate repository before sending a pull request so the feature can be discussed. This is to avoid you wasting your valuable time working on a feature that the project developers are not interested in accepting into the code base.

Fixing bugs

If you would like to fix a bug, please raise an issue in the appropriate repository before sending a pull request so it can be tracked.

Merge approval

The project maintainers use LGTM (Looks Good To Me) in comments on the code review to indicate acceptance. A change requires LGTMs from two of the maintainers of each component affected.

For a list of the maintainers, see the MAINTAINERS.md page in the appropriate repository.

3.10.2 Legal

Each source file must include a license header for the Apache Software License 2.0. Using the SPDX format is the simplest approach. e.g.

```
/*
Copyright <holder> All Rights Reserved.

SPDX-License-Identifier: Apache-2.0
*/
```

We have tried to make it as easy as possible to make contributions. This applies to how we handle the legal aspects of contribution. We use the same approach - the Developer's Certificate of Origin 1.1 (DCO) - that the Linux® Kernel community uses to manage code contributions.

We simply ask that when submitting a patch for review, the developer must include a sign-off statement in the commit message.

Here is an example Signed-off-by line, which indicates that the submitter accepts the DCO:

```
Signed-off-by: John Doe <john.doe@example.com>
```

You can include this automatically when you commit a change to your local git repository using the following command:

```
git commit -s
```

3.10.3 Communication

Please feel free to connect with us on our Slack channel or via email. Note that the projects in this repository are not formal products. As a result, the communication channels are to the maintainers and not to a support staff.

3.10.4 Setup

The documentation is a work in progress but should provide a good overview on how to get started with the project. The Dockerfile also provides a treasure trove of information on how to build the application, dependencies, and how to test the collector.

3.10.5 Testing

This project is in its infancy and with limited resources we haven't built many testers for the projects. For the sf-collector, we do have a set of unit tests that test the coverage of most of the events of interest in `sf-collector/tests`. These tests can be run using the bats testing framework. Directions on how to install bats are in the accompanied link. To run the tests, run `bats -t tests.bat` from the tests directory. Note, that the tests also rely on python3. Before conducting a pull request, these unit tests should be run. Note, there is a version of the docker image with a `testing` tag that contains bats and the unit tests. This might be useful for testing. Also, conducting a load test and running the application under valgrind is desirable for pull requests.

3.10.6 Coding style guidelines

We follow the LLVM Coding standards where possible across the projects. There is a .clang-format file in the master repo clang-format that can be used in conjunction with ClangFormat Tool to automatically format code. For linting, we use Clang Tidy Linter. This is referenced in the sf-collector Makefile.

3.11 Code of Conduct

3.12 Contributor Covenant Code of Conduct

3.12.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

3.12.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.12.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

3.12.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

3.12.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at Slack channel or via email. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

3.12.6 Attribution

This Code of Conduct is adapted from the Qiskit project's Code of Conduct and has roots from the Contributor Covenant, version 1.4, available at version.

3.13 Talks & Publications

If citing SysFlow, please use [TAS20].

Below you can find a complete list of talks and papers associated with SysFlow.

Note: Please reach out to us if you have an entry to add to this list.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[TAS20] Teryl Taylor, Frederico Araujo, and Xiaokui Shu. Towards an open format for scalable system telemetry. In *IEEE International Conference on Big Data (Big Data)*, 1031–1040. 2020. URL: https://arxiv.org/abs/2101.10474.

[BATJ24] William Blair, Frederico Araujo, Teryl Taylor, and Jiyong Jang. Automated synthesis of effect graph policies for microservice-aware stateful system call specialization. In *2024 IEEE Symposium on Security and Privacy (SP)*, 64–64. 2024. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00064.

[AT23] Frederico Araujo and Teryl Taylor. Relational observability for cloud-native security and data science. 2023. URL: https://sched.co/1K5IT.

[JAT22] Trent Jaeger, Frederico Araujo, and Teryl Taylor. Provenance tracking with attack graphs using sysflow. 2022. URL: https://avengercon.org/workshop/Provenance-Tracking-With-Attack-Graphs-Using-SysFlow/.

[AT22] Frederico Araujo and Teryl Taylor. Self-modulating endpoint observability. 2022. URL: https://sched.co/lDbn.

[SATJ21] Xiaokui Shu, Frederico Araujo, Teryl Taylor, and Jiyong Jang. An open stack for threat hunting in hybrid cloud with connected observability. 2021. URL: https://europe-arsenal-cfp.blackhat.com/.

[AT21] Frederico Araujo and Teryl Taylor. A pluggable edge-processing pipeline for SysFlow. 2021. URL: https://sched.co/ePsl.

[BATJ21] William Blair, Frederico Araujo, Teryl Taylor, and Jiyong Jang. Microservice-aware reference monitoring through hybrid program analysis. 2021. URL: https://sched.co/ePs3.

[AT20] Frederico Araujo and Teryl Taylor. SysFlow: scalable system telemetry for improved security analytics. 2020. URL: https://sched.co/VPW3.

PYTHON MODULE INDEX

S

INDEX

T

V