IPPS

|  |  |
| --- | --- |
| Version | 0.1 |
| Author | Tom Sumardi |
|  |  |

Sections:

<< TODO >>

Introduction:

This documentation intended to serve the purpose of giving detail level architecture description.

Convention/Requirements:

* Performance bound.
* IPPS operation will involve partial kernel and user space.

Summary:

IPPS pulls packets from SKB using pfring based on interface name within kernel space. First filtering being done on the packet layer by specifying libpcap filter syntax at the kernel level. After the packets are pulled, the packets go through series of filters on the user layer. A load balancer is also added to load balance the packets per flow based on 4 tuples (src/dst MAC address, src/dst IP address). For example, any interesting packets will be passed from one pipeline to the next from IPPS to PPP then to PHS. Any uninteresting packets packets are ignored. Future work might allow user level to tell kernel to ignore the flow.

Components:

* Configuration and registration thread
* IPPS master thread
* Packet Processing load balancer (worker) threads
* Packet L2-L4 processing and filtering
* Packet Payload Processing (PPP) plugins
* Packet Handling Service (PHS) plugins

Architecture:

<<<Figure1 here >>>

1. Component descriptions:
2. Configuration and registration thread:

Backend default main thread that is responsible for configuring and registering IPPS daemon, which uses AMQP as communication channel to the management server. This thread and IPPS master thread are the only threads that are up and running when the daemon first started. In the context of IPPS process, this thread communicates with IPPS master thread.

1. IPPS master thread:

Backend default main thread that spawns multiple worker threads based on configuration given. This thread will be idle until configuration and registration have been successfully applied. In the context of IPPS process, this thread communicates with Load balancer (worker) threads.

1. Load balancer (worker) threads:

The threads responsible for pulling packets from all the interfaces as fast as it can from the ring buffers. It utilizes pfring library in pulling the packets from kernel SKB and perform any necessary memory mapping the resource into the user space. Pfring library also responsible for segregating network interface, ring buffer creation, kernel-user space memory mapping, etc. The ring buffers are mapped to specific interface in one-to-many relationships with each ring buffer consuming packets per flow in 4 tuple relationships (src/dst IP address and src/dst MAC address).

<<<Figure2 here >>>

Figure 2. Example, n-thread instances are pulling packet from interface name “eth0”, which may have n-threads processing the packets with each thread assign its own stream flow. In case of network bonding, the interface will be broken down into each own separate interfaces.

1. Packet L2-L4 processing and filtering:

Packet L2-L4 processing and filtering functionality (context) will exist within each Load balancer (worker) thread. The idea is to pipeline incoming packets into a series of packet processing component and filters.

<<<Figure3 here >>>

As illustrated on figure 3, the packets are dissected and inspected from layer 2 up to layer 4 through a series of component plugins.

The filters will decide whether packets will be:

* + dropped,
  + moved to Packet Payload Processing (PPP) plugin or
  + moved to Packet Handling Service (PHS) plugins directly

Figure 4 (state machine)

above explanation can be summarized with figure 4 diagram of state machine.

1. Packet Payload Processing (PPP) plugin:

This section serves as brief introduction to PPP, please refer to specific documentation section for more detailed explanation. PP process packets above layer 4 or the payload, it dissects packets and perform regex matching based on specified configuration values

with HTTP protocol as default PPP plugin.

1. Packet Handling Service (PHS) plugins:

This section serves as brief introduction to PPP, please refer to specific documentation section for more detailed explanation. PHS is the last pipeline processing in packet life. For interested packets, it performs packet service operation if needed such as packet injection

in the case of transparent caching, packet forwarding in the case of HTTPS packet, etc in daisy chain manner.

1. Communication handshakes:

Ladder Diagram: TODO

- Daemon to Management Server (MS)

-> POST http://localhost:9494/api/v1/1/endpoint/register

'{"arguments":

{

"component": "ipps", "guid":"392846198346912"

}

}'

<-- OK {"sts":"success", "message":"success"}, {"sts": "error", "message":"resource unavailable"}

-> POST http://localhost:9494/api/v1/1/endpoint/configure

'{"arguments":

{

"component": "ipps", "guid":"392846198346912", "config":"{ }"

},

"config":"{ }"

}'

<-- OK {"sts":"success", "message":"success"}, {"sts": "error", "message":"resource unavailable"}

- Management Server (MS) to Daemon

-> POST http://localhost:9494/api/v1/1/endpoint/alive

'{"arguments":

{

"component": "ms", "guid":"56e56546646454"

}

}'

<-- OK {"sts":"success"}, {"sts": "error", "message":"resource unaivaiable"}