PF_RING User Guide

Linux High Speed Packet Capture

Version 1.1
January 2008

1. Introduction

PF_RING is a high speed packet capture library that turns a commodity PC into an efficient and cheap network measurement box suitable for both packet and active traffic analysis and manipulation. Moreover, PF_RING opens totally new markets as it enables the creation of efficient application such as traffic balancers or packet filters in a matter of lines of codes.

This manual is divided in two parts:

- PF_RING installation and configuration.
- PF_RING SDK.

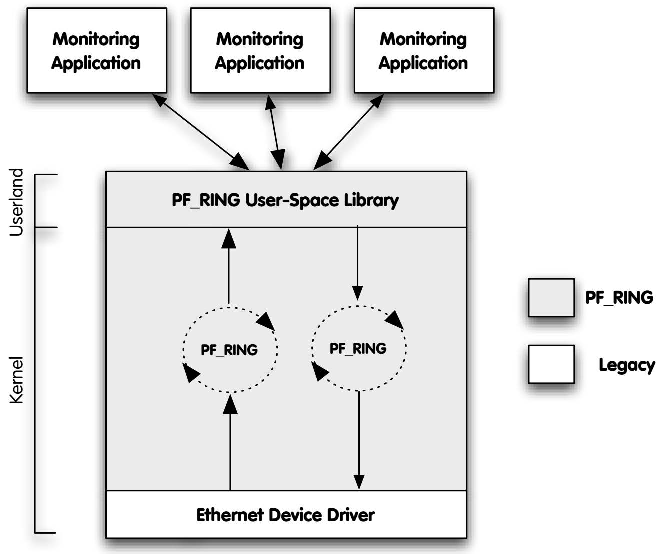
1.1 What's New with PF_RING?

- Release 1.0 (January 2008)
    - Initial PF_RING users guide.

- Release 1.1 (January 2008)
    - Described PF_RING plugins architecture.

2. PF_RING Installation

PF_RING's architecture is depicted in the figure below.



The main building blocks are:

- The accelerated kernel driver that provides low-level packet copying into the kernel PF_RINGs.
- The user space PF_RING SDK that provides transparent PF_RING-support to user-space applications.

When you download PF_RING you fetch the following components:

- An automatic patch mechanism allows you to automatically patch a vanilla kernel with PF_RING.
- The PF_RING user-space SDK.
- An enhanced version of the libpcap library that transparently takes advantage of PF_RING if installed, or fallback to the standard behavior if not installed.

PF_RING is downloaded by means of SVN as explained in http://www.ntop.org/PF_RING.html

2.1 Linux Kernel Installation

The PF_RING source code layout is the following:

```
 8 README          0 kernel/          32 mkpatch.sh     0 userland/
```

The Linux kernel patch is performed automatically by mkpatch.sh tool. This tool downloads from the Internet the linux kernel source and patches it. The patched kernel will be placed on a new directory named workspace that will sit at the same level of the other PF_RING files.

Users can decide what Linux kernel version to download by modifying the following mkpatch.sh variables:

```
VERSION=${VERSION:-2}
PATCHLEVEL=${PATCHLEVEL:-6}
SUBLEVEL=${SUBLEVEL:-18.4}
```

In the above configuration the kernel 2.6.18.4 will be downloaded.

After the kernel has been downloaded and patched, users need to compile and install the kernel as usual. Once the kernel is installed you need to modify your boot loader (usually lilo or grub) in order to let your system access the new kernel. Done this, you need to reboot the box and make sure you select the kernel you just installed as default kernel.

Note that:

- the kernel installation requires super user (root) capabilities.
- For some Linux distributions a kernel installation/compilation package is provided.

2.2 PF_RING Device Configuration

When PF_RING is activated, a new entry /proc/net/pf_ring is created.

```
nbox-factory:/home/deri# ls /proc/net/pf_ring/
info  plugins_info
nbox-factory:/home/deri# cd /proc/net/pf_ring/
nbox-factory:/proc/net/pf_ring# cat info
Version             : 3.7.5
Bucket length       : 2000 bytes
Ring slots          : 4096
Slot version        : 9
Capture TX          : Yes [RX+TX]
IP Defragment       : No
Transparent mode    : Yes
Total rings         : 0
Total plugins       : 2
nbox-factory:/proc/net/pf_ring# cat plugins_info
ID    Plugin
2     sip [SIP protocol analyzer]
12    rtp [RTP protocol analyzer]
```

PF_RING allows users to install plugins for handling custom traffic. Those plugins are also registered in the pf_ring /proc tree and can be listed by typing the plugins_info file.

2.3 Libpfring and Libpcap Installation

Both libpfring and libpcap are distributed in source format. They can be compiled as follows:

- cd userland/libpfring
- make
- sudo make install
- cd ../libpcap-0.9.7-ring/
- ./configure
- make

Note that the libpfring is reentrant hence it's necessary to link you PF_RING-enabled applications also against the -lpthread library.

IMPORTANT

Legacy pcap-based applications need to be recompiled against the new libpcap and linked with a PF_RING enabled libpcap.a in order to take advantage of PF_RING. Do not expect to use PF_RING without recompiling your existing application.

3. PF_RING for Application Developers

Conceptually PF_RING is a simple yet powerful technology that enables developers to create high-speed traffic monitor and manipulation applications in a small amount of time. This is because PF_RING shields the developer from inner kernel details that are handled by a library and kernel driver. This way developers can dramatically save development time focusing on they application they are developing without paying attention to the way packets are sent and received.

This chapter covers:

- The PF_RING API.
- Extensions to the libpcap library for supporting legacy applications.
- How to patch the Linux kernel for enabling PF_RING

3.1 The PF_RING API

The PF_RING internal data structures should be hidden to the user who can manipulate packets and devices only by means of the available API defined in the include file pfring.h that comes with PF_RING.

3.1.1 Return Codes

By convention, the library returns negative values for errors and exceptions. Non-negative codes indicate success.

3.1.2 PF_RING: Device Initialization

```
pfring* pfring_open(char *device_name, u_int8_t promisc, u_int8_t reentrant);
```

This call is used to initialize an PF_RING device hence obtain a handle of type struct pfring that can be used in subsequent calls. Note that:

- You can use both physical (e.g. eth0) and virtual (e.g. tap devices)
- You need super-user capabilities in order to open a device.

Input parameters:

device_name
Symbolic name of the PF_RING-aware device we're attempting to open (e.g. eth0).

promisc
If set to a value different than zero, the device is open in promiscuous mode.

reentrant
If set to a value different than zero, the device is open in reentrant mode. This is implemented by means of semaphores and it results is slightly worse performance. Use reentrant mode only for multithreaded applications.

Return value:

On success a handle is returned, NULL otherwise.

3.1.3 PF_RING: Device Termination

```
void pfring_close(pfring *ring);
```

This call is used to terminate an PF_RING device previously open. Note that you must always close a device before leaving an application. If unsure, you can close a device from a signal handler.

Input parameters:

ring
The PF_RING handle that we are attempting to close.

3.1.4 PF_RING: Read an Incoming Packet

```
int pfring_recv(pfring *ring, char* buffer, u_int buffer_len,  struct pfring_pkthdr *hdr,
                u_char wait_for_incoming_packet);
```

This call returns an incoming packet when available.

Input parameters:

ring
The PF_RING handle where we perform the check.

buffer
A memory area allocated by the caller where the incoming packet will be stored.

buffer_len
The length of the memory area above. Note that the incoming packet is cut if the incoming packet is too long for the allocated area.

hdr
A memory area where the packet header will be copied.

wait_for_incoming_packet
If 0 we simply check the packet availability, otherwise the call is blocked until a packet is available.

Return value:

The actual size of the incoming packet, from ethernet onwards.

3.1.5 PF_RING: Ring Clusters

int pfring_set_cluster(pfring *ring, u_int clusterId);

This call allows a ring to be added to a cluster that can spawn across address spaces. On a nuthsell when two or more sockets are clustered they share incoming packets that are balanced on a per-flow manner. This technique is useful for exploiting multicore systems of for sharing packets in the same address space across multiple threads.

Input parameters:

ring
The PF_RING handle to be cluster.

clusterId
A numeric identifier of the cluster to which the ring will be bound.

Return value:

Zero if success, a negative value otherwise.

int pfring_remove_from_cluster(pfring *ring);

This call allows a ring to be removed from a previous joined cluster.

Input parameters:

ring
The PF_RING handle to be cluster.

clusterId
A numeric identifier of the cluster to which the ring will be bound.

Return value:

Zero if success, a negative value otherwise.

3.1.6 PF_RING: Packet Reflection

int pfring_set_reflector(pfring *ring, char *reflectorDevice);

This call allows packets received from a ring not to be forwarded to user-space (as usual) but to be sent unmodified on a reflector device. This technique allows users to implement simple applications that set one or more filters and forward all packets matching the filter. All this is done in kernel space for maximum speed: the application just needs to instrument the ring without the need to fetch-and-forward packets.

Input parameters:

ring
The PF_RING handle to be used as reflector.

reflectorDevice
The reflector device (e.g. eth0). Note that it's not possible to use the same device for both receiving and forwarding packet.

Return value:

Zero if success, a negative value otherwise.

3.1.7 PF_RING: Packet Sampling

int pfring_set_sampling_rate(pfring *ring, u_int32_t rate /* 1 = no sampling */);

Implement packet sampling directly into the kernel. Note that this solution is much more efficient than implementing it in user-space. Sampled packets are only those that pass all filters (if any)

Input parameters:

ring
The PF_RING handle on which sampling is applied.

rate
The sampling rate. Rate of X means that 1 packet out of X is forwarded. This means that a sampling rate of 1 disables sampling

Return value:

Zero if success, a negative value otherwise.

3.1.8 PF_RING: Packet Filtering

PF_RING allows to filter packets in two ways: precise (a.k.a. hash filtering) or wildcard filtering. Precise filtering is used when it is necessary to track a precise 6-tuple connection <vlan Id, protocol, source IP, source port, destination IP, destination port>. Wildcard filtering is used instead whenever a filter can have wildcards on some of its fields (e.g. match all UDP packets regardless of their destination).

3.1.8.1 PF_RING: Wildcard Filtering

int pfring_add_filtering_rule(pfring *ring, filtering_rule* rule_to_add);

Add a filtering rule to an existing ring. Each rule will have a unique rule Id across the ring (i.e. two rings can have rules with the same id).

Input parameters:

ring
The PF_RING handle on which the rule will be added.

rule_to_add
The rule to add.

Return value:

Zero if success, a negative value otherwise.

int pfring_remove_filtering_rule(pfring *ring, u_int16_t rule_id);

Remove a previously added filtering rule.

Input parameters:

ring
The PF_RING handle on which the rule will be added.

rule_id
The id of a previously added rule that will be removed.

Return value:

Zero if success, a negative value otherwise (e.g. the rule does not exist).

int pfring_get_filtering_rule_stats(pfring *ring, u_int16_t rule_id, char* stats, u_int *stats_len);

Read statistics of a hash filtering rule.

Input parameters:

ring
The PF_RING handle from which stats will be read.

rule_id
The rule id that identifies the rule for which stats are read.

stats
A buffer allocated by the user that will contain the rule statistics. Please make sure that the buffer is large enough to contain the statistics.

stats_len
The size (in bytes) of the stats buffer.

Return value:

Zero if success, a negative value otherwise (e.g. the rule does not exist).

3.1.8.2 PF_RING: Hash Filtering

```
int pfring_handle_hash_filtering_rule(pfring *ring,
                                      hash_filtering_rule* rule_to_add,
                                      u_char add_rule);
```

Add or remove a hash filtering rule.

Input parameters:

ring
The PF_RING handle from which stats will be read.

rule_to_add
The rule that will be added/removed.

add_rule
If set to a positive value the rule is added, if zero the rule is removed

Return value:

Zero if success, a negative value otherwise (e.g. the rule to be removed does not exist).

```
int pfring_get_hash_filtering_rule_stats(pfring *ring,
                                         hash_filtering_rule* rule,
                                         char* stats, u_int *stats_len);
```

Read statistics of a hash filtering rule.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

rule
The rule for which stats are read. This needs to be the same rule that has been previously added.

stats
A buffer allocated by the user that will contain the rule statistics. Please make sure that the buffer is large enough to contain the statistics.

stats_len
The size (in bytes) of the stats buffer.

Return value:

Zero if success, a negative value otherwise (e.g. the rule to be removed does not exist).

3.1.8.3 PF_RING: Filtering Policy

int pfring_toggle_filtering_policy(pfring *ring, u_int8_t rules_default_accept_policy);

Set the default filtering policy. This means that if no rule is matching the incoming packet the default policy will decide if the packet is forwarded to user space of dropped. Note that filtering rules are limited to a ring, so each ring can have a different set of rules and default policy.

Input parameters:

ring
The PF_RING handle on which the rule will be added/removed.

rules_default_accept_policy
If set to a positive value the default policy is accept (i.e. forward packets to user space), drop otherwise

Return value:

Zero if success, a negative value otherwise.

3.1.9 PF_RING: Miscellaneous Functions

int pfring_enable_ring(pfring *ring);

A ring is not enabled (i.e. incoming packets are dropped) until the user space application calls pfring_recv() or the above function. This function should usually not be called unless the user space application sets drop-filters and periodically reads statistics from the ring.

Input parameters:

ring
The PF_RING handle to enable.

Return value:

Zero if success, a negative value otherwise.

int pfring_stats(pfring *ring, pfring_stat *stats);

Read ring statistics (packets received and dropped).

Input parameters:

ring
The PF_RING handle to enable.

stats
A user-allocated buffer on which stats will be stored.

Return value:

Zero if success, a negative value otherwise.

int pfring_version(pfring *ring, u_int32_t *version);

Read the ring version. Note that is the ring version is 3.7 the retuned ring version is 0x030700.

Input parameters:

ring
The PF_RING handle to enable.

version
A user-allocated buffer on which ring version will be copied.

Return value:

Zero if success, a negative value otherwise.

3.2 The C++ PF_RING interface

The C++ interface (see. PF_RING/userland/libpfring/c++/) is equivalent to the C interface. No major changes have been made and all the methods have the same name as C. For instance:

- C: int pfring_stats(pfring *ring, pfring_stat *stats);
- C++: inline int get_stats(pfring_stat *stats);

4. Writing PF_RING Plugins

Since version 3.7, developers can write plugins in order to delegate to PF_RING activities like:

- Packet payload parsing
- Packet content filtering
- In-kernel traffic statistics computation.

In order to clarify the concept, imagine that you need to develop an application for VoIP traffic monitoring. In this case it's necessary to:

- parse signaling packets (e.g. SIP or IAX) so that those that only packets belonging to interesting peers are forwarded.
- compute voice statistics into PF_RING and report to user space only the statistics, not the packets.

In this case a developer can code two plugins so that PF_RING can be used as an advanced traffic filter and a way to speed-up packet processing by avoiding packets to cross the kernel boundaries when not needed.

The rest of the chapter explains how to implement a plugin and how to call it from user space.

4.1 Implementing a PF_RING Plugin

Inside the directory kernel/net/ring/plugins/ there is a simple plugin called dummy_plugin that shows how to implement a simple plugin. Let's explore the code.

Each plugin is implemented as a Linux kernel module. Each module must have two entry points, module_init and module_exit, that are called when the module is insert and removed. The module_init function, in the dummy_plugin example it's implement by the function dummy_plugin_init(), is responsible for registering the plugin by calling the do_register_pfring_plugin() function. The parameter passed to the registration function is a data structure of type 'struct pfring_plugin_registration' that contains:

- a unique integer pluginId.
- pfring_plugin_handle_skb: a pointer to a function called whenever an incoming packet is received.
- pfring_plugin_filter_skb: a pointer to a function called whenever a packet needs to be filtered. This function is called after pfring_plugin_handle_skb().
- pfring_plugin_get_stats: a pointer to a function called whenever a user wants to read statistics from a filtering rule that has set this plugin as action.

A developer can choose not to implement all the above functions, but in this case the plugin will be limited in functionality (e.g. if pfring_plugin_filter_skb is set to NULL filtering is not supported).

4.1.1 PF_RING Plugin: Handle Incoming Packets

```
static int plugin_handle_skb(filtering_rule_element *rule,
                             filtering_hash_bucket *hash_rule,
                             struct pcap_pkthdr *hdr,
                             struct sk_buff *skb,
                             u_int16_t filter_plugin_id,
                             struct parse_buffer *filter_rule_memory_storage);
```

This function is called whenever an incoming packet (RX or TX) is received. This function typically updates rule statistics. Note that if the developer has set this plugin as filter plugin, then the packet has:

- already been parsed
- passed a rule payload filter (if set).

Input parameters:

rule
A pointer to a wildcard rule (if this plugin has been set on a wildcard rule) or NULL (if this plugin has been set to a hash rule).

hash_rule
A pointer to a hash rule (if this plugin has been set on a hash rule) or NULL (if this plugin has been set to a wildcard rule). Note if rule is NULL, hash_rule is not, and vice-versa.

hdr
A pointer to a pcap packet header for the received packet. Please note that:

- the packet is already parsed
- the header is an extended pcap header containing parsed packet header metadata.

skb
A sk_buff datastructure used in Linux to carry packets inside the kernel.

filter_plugin_id
The id of the plugin that has parsed packet payload (not header that is already stored into hdr). if the filter_plugin_id is the same as the id of the dummy_plugin then this packet has already been parsed by this plugin and the parameter filter_rule_memory_storage points to the payload parsed memory.

filter_rule_memory_storage
Pointer to a data structure containing parsed packet payload information that has been parsed by the plugin identified by the parameter filter_plugin_id. Note that:

- only one plugin can parse a packet.
- the parsed memory is allocated dynamically (i.e. via kmalloc) by plugin_filter_skb and freed by the PF_RING core.

Return value:

Zero if success, a negative value otherwise.

4.1.2 PF_RING Plugin: Filter Incoming Packets

```
int plugin_filter_skb(filtering_rule_element *rule,
                      struct pcap_pkthdr *hdr,
                      struct sk_buff *skb,
                      struct parse_buffer **parse_memory)
```

This function is called whenever a previously parsed packet (via plugin_handle_skb) incoming packet (RX or TX) needs to be filtered. In this case the packet is parsed, parsed information is returned and the return value indicates whether the packet has passed the filter.

Input parameters:

rule
A pointer to a wildcard rule that contains a payload filter to apply to the packet.

hdr
A pointer to a pcap packet header for the received packet. Please note that:

- the packet is already parsed
- the header is an extended pcap header containing parsed packet header metadata.

skb
A sk_buff data structure used in Linux to carry packets inside the kernel.

Output parameters:

parse_memory
A pointer to a memory area allocated by the function, that will contain information about the parsed packet payload.

Return value:

Zero if the packet has not matched the rule filter, a positive value otherwise.

4.1.3 PF_RING Plugin: Read Packet Statistics

```
int plugin_plugin_get_stats(filtering_rule_element *rule,
                            filtering_hash_bucket  *hash_bucket,
                            u_char* stats_buffer,
                            u_int stats_buffer_len)
```

This function is called whenever a user space application wants to read statics about a filtering rule.

Input parameters:

rule
A pointer to a wildcard rule (if this plugin has been set on a wildcard rule) or NULL (if this plugin has been set to a hash rule).

hash_rule
A pointer to a hash rule (if this plugin has been set on a hash rule) or NULL (if this plugin has been set to a wildcard rule). Note if rule is NULL, hash_rule is not, and vice-versa.

stats_buffer
A pointer to a buffer where statistics will be copied..

stats_buffer_len
Length in bytes of the stats_buffer.

Return value:

The length of the rule stats, or zero in case of error.

4.2 Using a PF_RING Plugin

A PF_RING based application, can take advantage of plugins when filtering rules are set. The filtering_rule data structure is used to both set a rule and specify a plugin associated to it.

```
filtering_rule rule;

rule.rule_id = X;
....
rule.plugin_action.plugin_id = MY_PLUGIN_ID;
```

When the plugin_action.plugin_id is set, whenever a packet matches the header portion of the rule, then the MY_PLUGIN_ID plugin (if registered) is called and the plugin_filter_skb () and plugin_handle_skb() are called.

If the developer is willing to filter a packet before plugin_handle_skb() is called, then extra filtering_rule fields need to be set. For instance suppose to implement a SIP filter plugin and to instrument it so that only the packets with INVITE are returned. The following lines of code show how to do this.

```
struct sip_filter *filter = (struct sip_filter*)rule.extended_fields.filter_plugin_data;

rule.extended_fields.filter_plugin_id = SIP_PLUGIN_ID;
filter->method = method_invite;
filter->caller[0]  = '\0'; /* Any caller */
filter->called[0]  = '\0'; /* Any called */
filter->call_id[0] = '\0'; /* Any call-id */
```

As explained before, the pfring_add_filtering_rule() function is used to register filtering rules.