I Do See

SURICATA

Mixing IPS/IDS Mode

November 4, 2015

About me

- **Software developer @ Stamus Networks**
- Ntop
- EmergingThreats

- Suricata developer
- Netfilter developer

Agenda

1. Introduction
2. IPS
3. Nfqueue
4. Nflog
5. Mixed mode
6. Implementation
7. Demonstration
8. Conclusion

IPS MODE

We have different ways to setup the IPS mode :

- NFQUEUE : Use Netfilter on Linux
- IPFW : Use divert socket on FreeBSD
- AF_PACKET : Level 2 software bridge

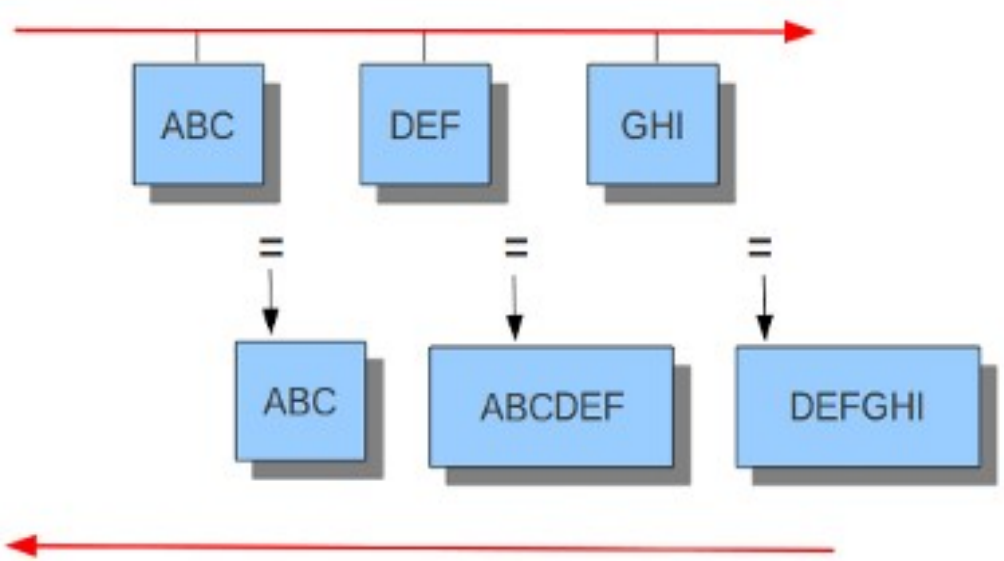
IPS MODE

How the IPS mode works in Suricata :

- It inspects packets immediately before sending them to the receiver
- The packets are inspected using the sliding window concept :
  - It inspects data as they come in until the tcp connection is closed

IPS MODE

Sliding Window :



Sliding window = 6

1. Suricata gets the first chunk and inspect it.

2. Then gets the second chunk, put it together with the first, and inspect it.

3. At the end, gets the third chunk, cut off the first one, put together second chunk with the third, and inspect it.

IPS MODE

On the administrative side, we must have signatures with a proper action in our ruleset.

An « action » is a property of the signature which determines what will happen when a signature matches the incoming data.

IPS MODE

In IDS mode, we have the following actions :

- Pass
  - Suricata stops scanning the packet and skips to the end of all rules (only for this packet)
- Alert
  - Suricata fires up an alert for the packet matched by a signature

IPS MODE

In IPS mode, we have two actions available to block network traffic : drop and reject.

- Drop:
  - If a signature containing a drop action matches a packet, this is discarded immediately and won't be sent any further.
  - The receiver doesn't receive a message, resulting in a time-out.
  - All subsequent packets of a flow are dropped
  - Suricata generates an alert for this packet.
  - This only concerns the IPS mode

IPS MODE

- Reject
  - This is an active rejection of the packet, both receiver and sender receive a reject packet.
  - If the packet concerns TCP, it will be a reset-packet, otherwise it will be an ICMP-error packet for all other protocols.
  - Suricata generates an alert too.
  - In IPS mode, the packet will be dropped as in the drop action
  - Reject in IDS mode is called IDPS

IPS MODE

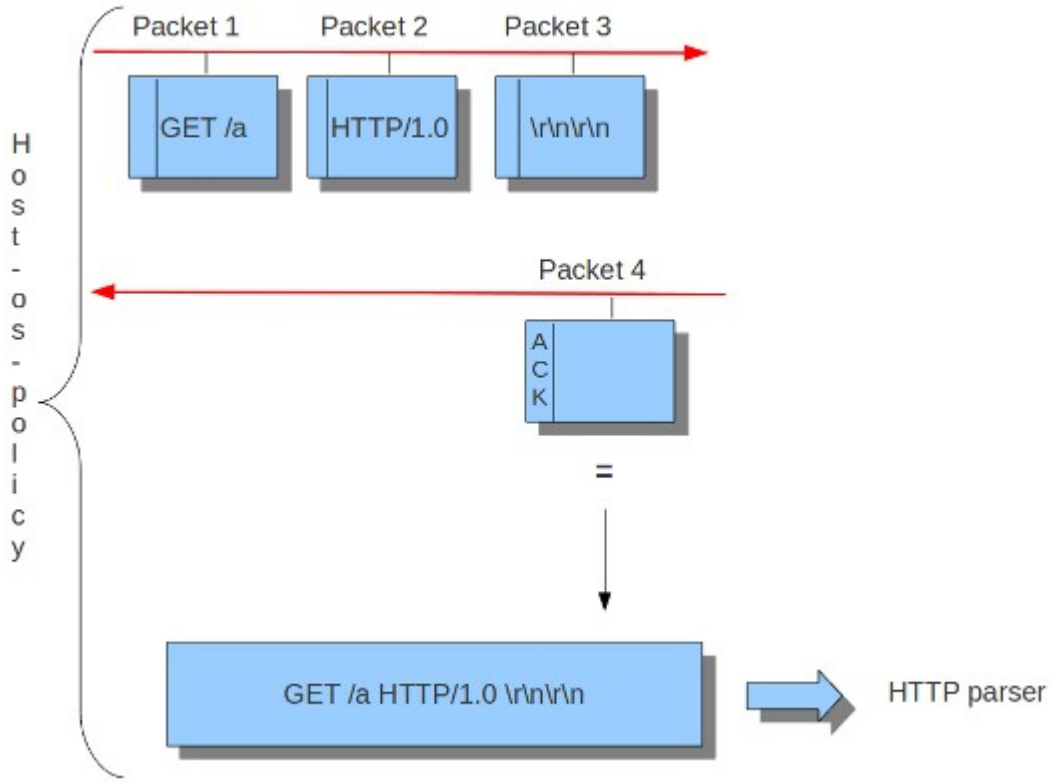
Working in IPS mode, the behavior of Suricata is influenced by stream.inline setting :

Normally, we analyse data once we know they have been received by the receiver, in term of TCP this means after it has been ACKed.

But in IPS it does not work like this, because the data have reached the host that we protect.

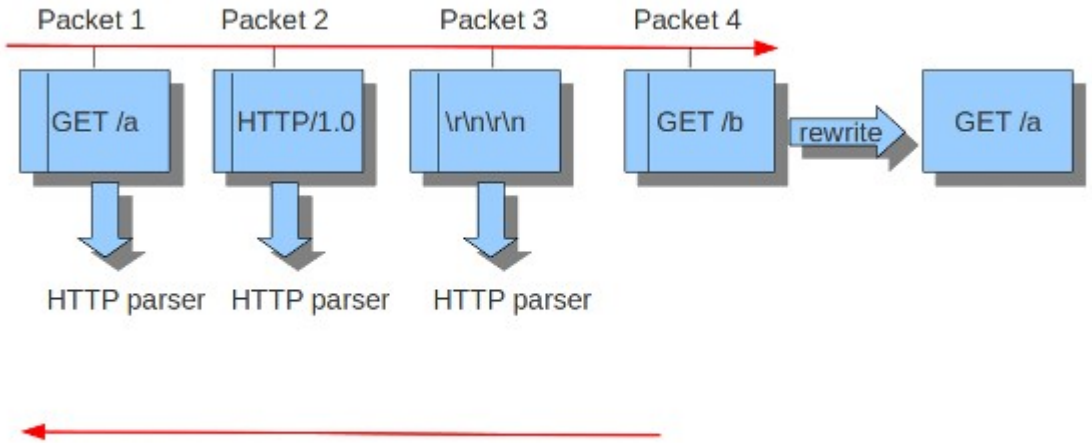
IPS MODE

Suricata reassembles on ACKed data then inspects them :



IPS MODE

Enabling stream.inline permits us to analyze data before they have been ACKed:



When Suricata receives a packet, it triggers the reassembly process itself.

In this case, if the detection engine decides a drop is required, the packet containing the data itself can be dropped, not just the ACK.

As a consequence of inline mode, Suricata can drop or modify packets if stream reassembly requires it.

NETFILTER

Netfilter is a framework, developed by Netfilter Organization, inside the Linux kernel that enables packet filtering, network address translation, and other packet mangling.

Nftables is the new netfilter project that aims to replace the existing {ip,ip6,arp,ebt}tables tools.

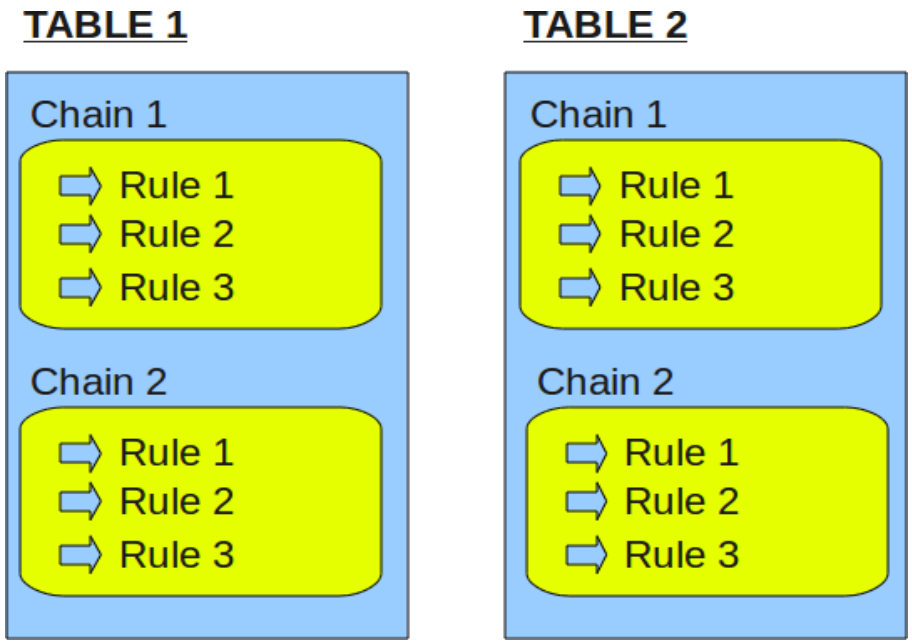
It provides a new packet filtering framework, a new user-space tool (nft) and a compatibility layer for old tools.

It's time to move on nftables !

NETFILTER

When a packet enters the firewall, it starts to go through a series of steps in the kernel, before it is sent to the correct application, or forwarded to another host, or whatever happens to it.

These steps are called **rules**, and are organized hierarchically in **tables** and **chains**.



NETFILTER

- Tables :
  - Each table has a specific purpose and chains
  - There are 5 main built-in tables in iptables
  - In nftables, all tables are user-defined

- Chains :
  - Each chain has a specific purpose and contains a ruleset that is applied on packets that traverse the chain

NETFILTER : FILTER

FILTER table :

It is used for filtering packets.

We can match packets and filter them in whatever way we may want.

This is the place that we actually take actions against packets.

For example, we could DROP, LOG, ACCEPT or REJECT packets.

There are three chains built-in to this table.

NETFILTER : FILTER

- INPUT : it's used on all packets that are destined for the firewall
- FORWARD : it's used on all non-locally generated packets that are not destined for our localhost
- OUTPUT : it's used for all locally generated packets

NETFILTER : NAT

NAT table :

It's used mainly for Network Address Translation.

NATed packets get their IP addresses (or ports) altered, according to our rules.

Packets in a stream only traverse this table once.

We assume that the first packet of a stream is allowed.

The rest of the packets in the same stream are automatically NATed, or Masqueraded, etc.

NETFILTER : NAT

- PREROUTING : It's used to alter packets as soon as they get into the firewall
- OUTPUT : it's used for altering locally generated packets before they get to the routing decision
- POSTROUTING : it's used to alter packets just as they are about to leave the firewall

NETFILTER : MANGLE

MANGLE table :

This table is used mainly for mangling packets.

Among other things, we can change the contents of different packets and some of their headers.

For example : TTL, TOS or MARK.

NETFILTER : MANGLE

- PREROUTING : it's used for altering packets just as they enter the firewall and before they hit the routing decision

- POSTROUTING : it's used to mangle packets just after all routing decisions have been made

- INPUT : it's used to alter packets after they have been routed to the local host itself, but before the userspace software sees the data

- FORWARD : it's used to mangle packets after they have hit the first routing decision, but before they actually hit the last routing decision

- OUTPUT : it's used for altering locally generated packets after they enter the routing decision

NETFILTER : RAW

RAW table :

This table and its chains are used before any other tables in netfilter.

- PREROUTING : it's used for all incoming packets
- OUTPUT : it's used to alter the locally generated packets before they hit any of the other netfilter subsystem

NETFILTER : RULE

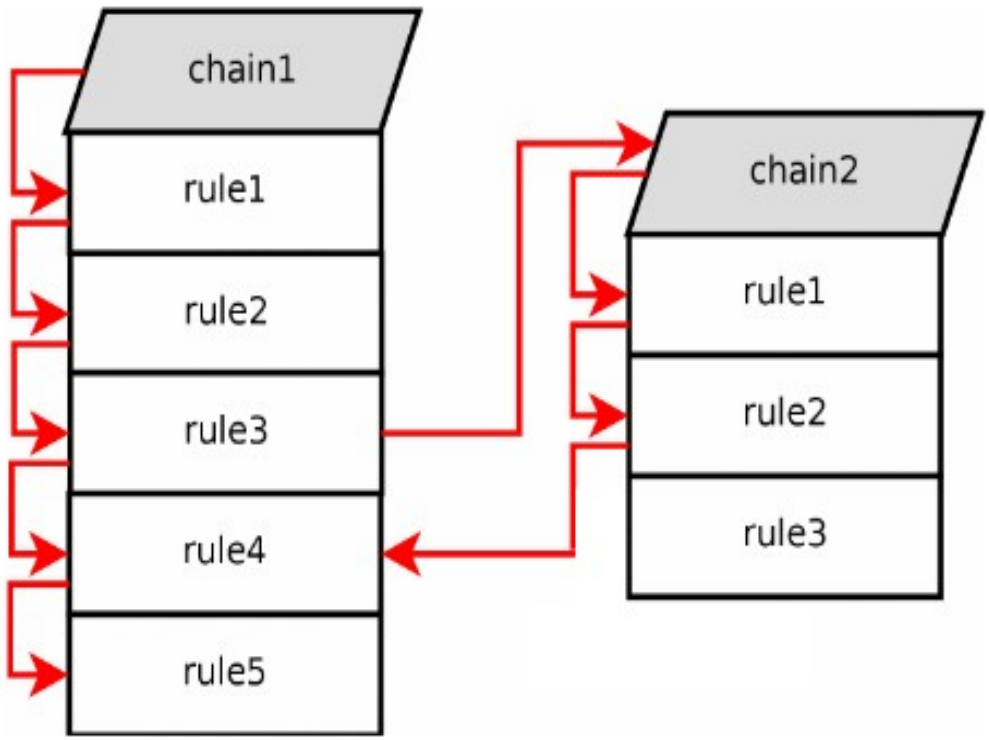
A **rule** is a set of criteria with a target that specify the action to take.

Target :

- ACCEPT : the packet is accepted (it's sent to the destination)
- DROP : the packet is dropped (it's not sent to the destination)
- User-defined Chain : another ruleset is executed
- RETURN : stops executing the next set of rules in the current chain for this packet. The control will be returned to the calling chain.

NETFILTER : RETURN TARGET

RETURN target :



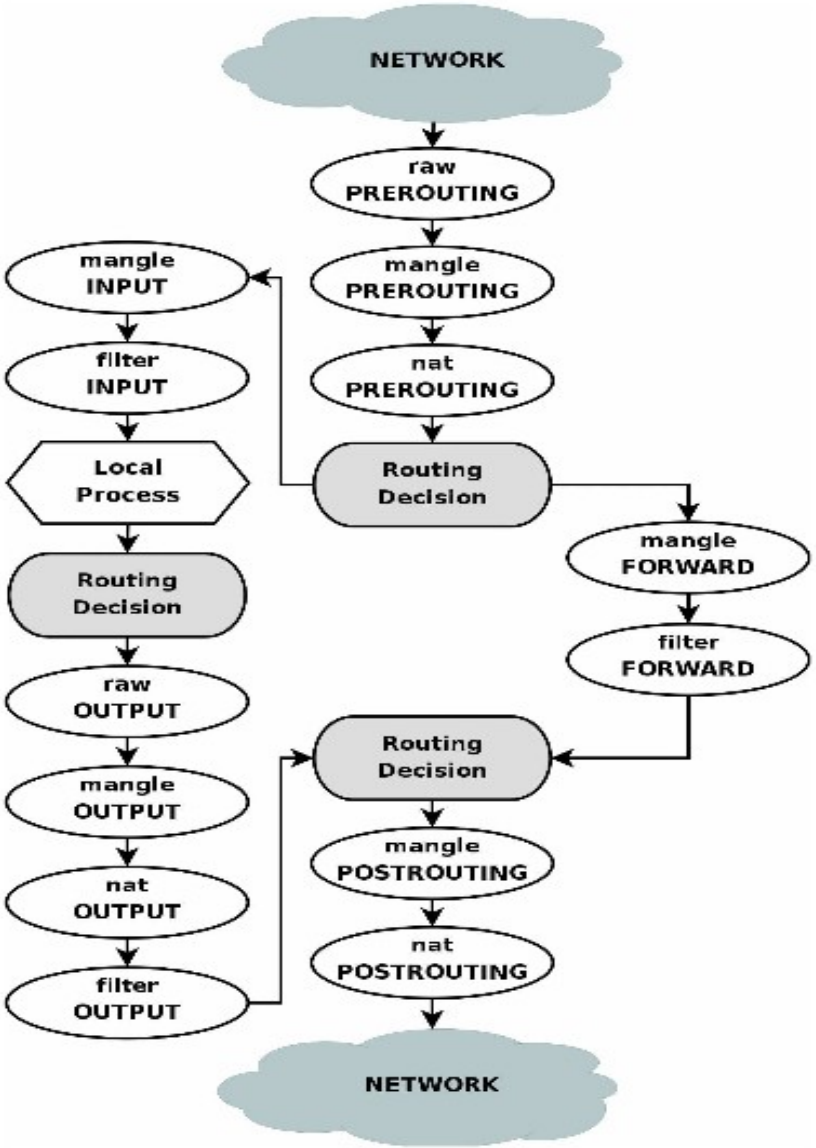
1. A packet traverses chain1

2. When rule3 matches the packet, it is sent to chain2

3. The packet traverses chain2 until is matched by rule2

4. At this point, packet returns to chain1 and rule3 is not tested

NETFILTER : PACKET PATH



NFQUEUE

- It is used in Suricata to work in IPS mode, performing actions like DROP or ACCEPT on the packets.

- With NFQUEUE we are able to delegate the verdict on the packet to a userspace software.

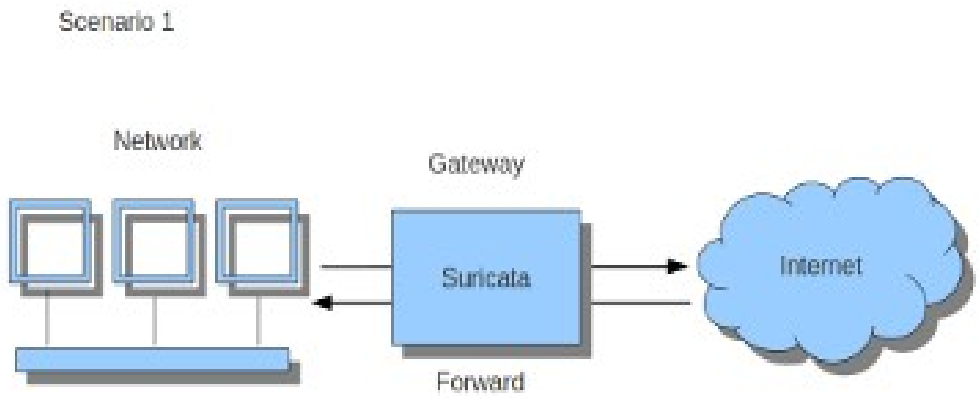
- The following rules will ask a userspace software connected to queue 0 for a decision.

nft add filter forward queue num 0

iptables -A FORWARD -j NFQUEUE --queue-num 0

NFQUEUE

These rules mean that we are in the following scenario :

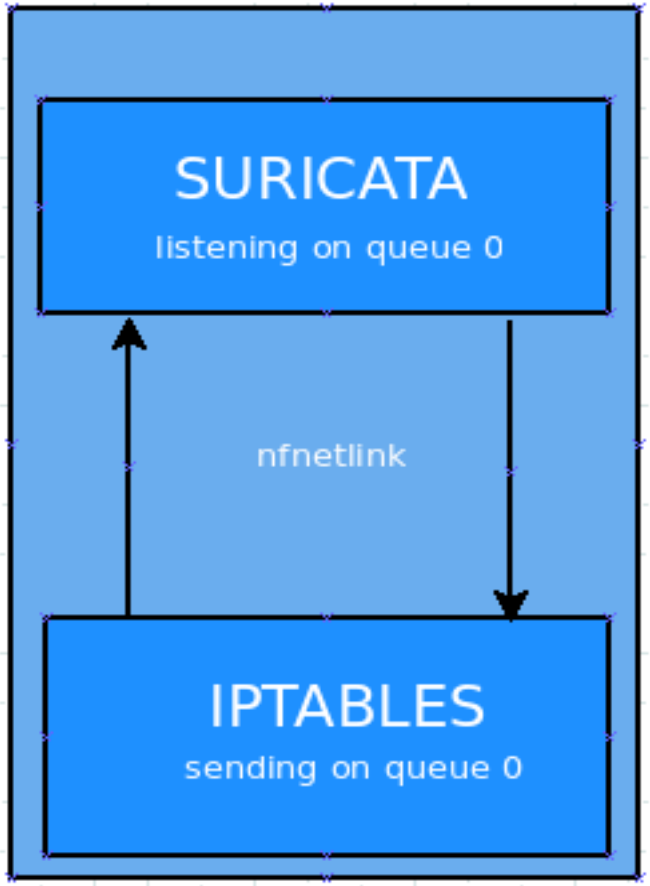


In this case, Suricata is running on a gateway and is meant to protect the computers behind that gateway.

NFQUEUE

The following picture explains how NFQUEUE works with Suricata in IPS mode :

1. Incoming packet matched by a rule is sent to Suricata through nfnetlink
2. Suricata receives the packet and issues a verdict depending on our ruleset
3. The packet is either transmitted or rejected by kernel



NFQUEUE

The following options can be set with NFQUEUE :

--queue-num : queue number

--queue-balance : packet is queued by the same rules to multiple queues which are load balanced

--queue-bypass : packet is accepted when no software is listening to the queue

fail-open : packet is accepted when queue is full

batching verdict : a verdict is sent to all packets

NFQUEUE

NFQUEUE number of packets per second on a single queue is limited due to the nature of nfnetlink communication.

Batching verdict can help but without an efficient improvement.

Starting Suricata with multiple queue could improve it:

« suricata -q 0 -q 1 -c /etc/suricata.yaml »

NFLOG

- NFLOG is for LOGging

- It is similar to NFQUEUE but it only sends a copy of packet without issuing a verdict.

- The communication between NFLOG and a userspace software is made through netlink socket.

- In fact, the following rules will send all packets to netlink socket 10 :

nft add rule filter input ip log group 10

iptables -A INPUT -j NFLOG --nflog-group 10

NFLOG

- It's used to log packet by the kernel packet filter and pass it via userspace software.

- Some software that uses NFLOG :
  - Ulogd2 (Netfilter userspace logging daemon)
  - Wireshark

NFLOG

--nflog-group : number of the netlink multicast group

--nflog-range <N> : number of bytes up to which the packet is copied

--nflog-threshold : if a packet is matched by a rule, and already n packets are in the queue, the queue is flushed to userspace

--nflog-prefix : string associated with every packet logged

MIXED MODE : INTRO

We are ready to talk about the mixed mode.

Let's start with an easy question :

What is the mixed mode ?

It's a feature that permits to run different runmodes, giving us the possibility to mix different capture methods, like NFQUEUE and NFLOG.

The key point of mixed mode is the fact you decide on a per packet basis if it is IDS or inline stream handling.

MIXED MODE : MOTIVATION

This mode gives us two advantages :

1. Having a mixed environment

- We may want to block some traffic, and inspect some.

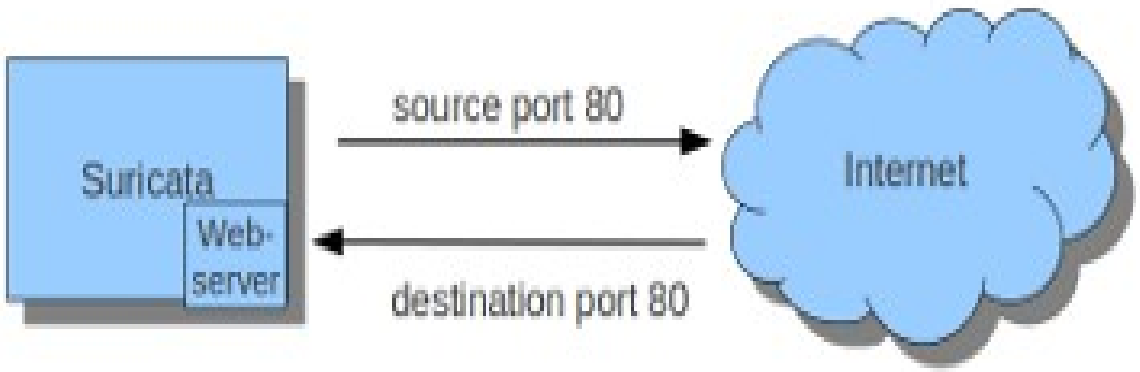
2. Technical simplification

- We could have an IPS/IDS system, as mixed mode, running many suricata instances with different configuration files

MIXED MODE : USAGE

The usage, although it requires some steps, is quite simple :

- Web server on 80: can't block trafic
- Rest of trafic is less sensitive



MIXED MODE : USAGE

1. Add iptables rules with NFQ/NFLOG target :

we want to be sure not to cut off a webserver,
but we want to inspect port 80

Send other than 80 to IPS

- nft add rule filter forward tcp dport not 80 queue num 0

  # Port 80 to IDS

- nft add rule filter forward tcp dport 80 log group 2

Iptables way :

- iptables -A FORWARD -p tcp ! --dport 80 -j NFQUEUE
- iptables -A FORWARD -p tcp –dport 80 -j NFLOG --nflog-group 2

MIXED MODE : USAGE

2. We may want to modify suricata config file.

In this case, we could modify nflog or nfq because we are using them as examples.

But it depends on the runmode you choice.

```
#nflog support
nflog:
    # netlink multicast group
    # (the same as the iptables --nflog-group param)
    # Group 0 is used by the kernel, so you can't use it
  - group: 2
    # netlink buffer size
    buffer-size: 18432
    # put default value here
  - group: default
    # set number of packet to queue inside kernel
    qthreshold: 1
    # set the delay before flushing packet in the queue inside kernel
    qtimeout: 100
    # netlink max buffer size
    max-size: 20000
```

MIXED MODE : USAGE

3. Run suricata :

« suricata -c suricata.yaml -q 0 –nflog -v »

As we said, it's quite simple...

but, let's enhance our skills !

MIXED MODE : NINJA USAGE

We could also send all the traffic of a suspicious IP from IDS mode to IPS : how ?

Let's suppose that we notice a suspicious IP in the eve log file and we want to block it :


```
{
  "timestamp": "2004-05-13T12:17:12.328438+0200",
  "flow_id": 41969728,
  "pcap_cnt": 39,
  "event_type": "http",
  "src_ip": "145.254.160.237",
  "src_port": 3372,
  "dest_ip": "65.208.228.223",
  "dest_port": 80,
  "proto": "TCP",
  "tx_id": 0,
  "http": {
    "hostname": "www.ethereal.com",
    "url": "/download.html",
    "http_user_agent": "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6) Gecko/20040113",
    "http_content_type": "text/html",
    "http_refer": "http://www.ethereal.com/development.html",
    "http_method": "GET",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 18070
  }
}
```

MIXED MODE : NINJA USAGE

We should add a rule to block the incoming traffic from this IP :

- nft add rule filter input ip saddr 145.254.160.237 queue 0

But this solution is not very performing, because if we want to block another IP we need to add another identical rule

=> rules duplication

MIXED MODE : NINJA

A better solution is to build a set containing all suspiscious IPs and block all incoming traffic from them :

- nft add set filter suspisciousips { type ipv4_addr\;}
- nft add element filter suspisciousip { 145.254.160.237 }
- nft add rule ip input ip saddr @suspiscious queue 0

MIXED MODE : NINJA USAGE

- iptables way :

  - ipset create suspiciousip
  - ipset add suspiciousip 145.254.160.237

  - iptables -I FORWARD -m set –set suspiciousip -j NFQUEUE –queue-num 0

MIXED MODE : NINJA USAGE

if we are real ninja, we can automate things better ! (see later)

MIXED MODE : IMPLEMENTATION

1. Store multiple runmodes
2. Accept many runmodes from the command line
3. Set a runmode id to the interfaces
4. Figure out how to handle a packet

RUNNING MODE

A running mode specifies what Suricata will have to do : capture packet, run tests, offline analysis, etc.

```
/* Run mode */
enum RunModes {
    RUNMODE_UNKNOWN = 0,
    RUNMODE_PCAP_DEV,
    RUNMODE_PCAP_FILE,
    RUNMODE_PFRING,
    RUNMODE_NFQ,
    RUNMODE_NFLOG,
    RUNMODE_IPFW,
    RUNMODE_ERF_FILE,
    RUNMODE_DAG,
    RUNMODE_AFP_DEV,
    RUNMODE_NETMAP,
    RUNMODE_TILERA_MPIPE,
    RUNMODE_UNITTEST,
    RUNMODE_NAPATECH,
    RUNMODE_UNIX_SOCKET,
    RUNMODE_USER_MAX, /* Last standard running mode */
    RUNMODE_LIST_KEYWORDS,
    RUNMODE_LIST_APP_LAYERS,
    RUNMODE_LIST_CUDA_CARDS,
    RUNMODE_LIST_RUNMODES,
    RUNMODE_PRINT_VERSION,
    RUNMODE_PRINT_BUILDINFO,
    RUNMODE_PRINT_USAGE,
    RUNMODE_DUMP_CONFIG,
    RUNMODE_CONF_TEST,
    RUNMODE_LIST_UNITTEST,
    RUNMODE_ENGINE_ANALYSIS,
#ifdef OS_WIN32
    RUNMODE_INSTALL_SERVICE,
    RUNMODE_REMOVE_SERVICE,
    RUNMODE_CHANGE_SERVICE_PARAMS,
#endif
    RUNMODE_MAX,
};
```

MIXED MODE : STEP 1

1. Store multiple runmodes

Suricata stores the runmode in the following variable :

- int run_mode ;

This has been replaced by the following structure :

+ typedef struct RunModeList_ {

+ int run_mode[RUNMODES_MAX] ;

+ int runmodes_cnt ;

+ } ;

MIXED MODE : STEP 1

- The runmodes are stored into a two slot array.
  - (+ #define RUNMODES_MAX 2)
- Not all runmodes can be mixed
  - (doesn't make sense to run UTs and capture packets)

- Currently only AF_PACKET, NFQ, NFLOG can be mixed
  - The second slot is activated when one of these capture method is chosen

- The second slot can store only runmodes that represent capture packets methods.
  - (For example, the runmode for UT will be always stored in the first slot)

MIXED MODE : STEP 2

2. Accept many runmodes from the command line

It was possible to specify only one runmode :

We can specify at most two runmodes now.

MIXED MODE : STEP 2

- It's forbidden to run more than two runmodes :

  + if (runmodes->runmodes_cnt < RUNMODES_MAX) {

  + runmodes->runmodes_cnt++;

  + }

- It's used also to check which runmodes can be mixed

  + if (runmodes->run_mode[runmodes->runmodes_cnt] == RUNMODE_UNKNOWN && runmodes->runmodes_cnt == 0)

MIXED MODE : STEP 3

3. Set a runmode to the interfaces

Suricata stores the interfaces specified with a runmode in a list, and they will be used to set-up the capture threads.

If only one runmode is selected we'll have interfaces belonging to the same runmode even if it is not specified.

MIXED MODE : STEP 3

Since we have many runmodes now, we don't know which runmode the interface belongs to.

Look at the following example.

MIXED MODE : STEP 3

If we start suricata in mixed mode :

« suricata -c suri.yaml -q 0 –nflog -v »

What happens is :

```
[8300] 27/10/2015 -- 09:07:22 - (util-device.c:160) <Info> (LiveBuildDeviceListCustom) -- Adding group 2 from config file
[8300] 27/10/2015 -- 09:07:22 - (util-runmodes.c:189) <Info> (RunModeSetLiveCaptureAutoFp) -- Using 2 live device(s).
[8300] 27/10/2015 -- 09:07:22 - (runmode-nflog.c:86) <Info> (ParseNflogConfig) -- Parsing group 0

Program received signal SIGSEGV, Segmentation fault.
ConfNodeLookupChild (node=node@entry=0x0, name=name@entry=0x5e0f90 "buffer-size") at conf.c:725
725         TAILQ_FOREACH(child, &node->head, next) {
(gdb)
```

MIXED MODE : STEP 3

NFLOG is trying to parse a group with id 0, but zero identifies a queue number, not a nflog group.

Although NFLOG group and NFQUEUE queue num are not really interfaces, they are handled as such !

MIXED MODE : STEP 3

The solution is quite simple, we specify the runmode id of the interface :

```
diff --git a/src/util-device.h b/src/util-device.h
index fd6a821..08b9235 100644
--- a/src/util-device.h
+++ b/src/util-device.h
@@ -20,10 +20,12 @@

 #include "queue.h"
 #include "unix-manager.h"
+#include "runmodes.h"

 /** storage for live device names */
 typedef struct LiveDevice_ {
     char *dev;  /**< the device (e.g. "eth0") */
+    enum RunModes runmode; /**< the runmode (e.g. "RUNMODE_NFLOG") */
     int ignore_checksum;
     SC_ATOMIC_DECLARE(uint64_t, pkts);
     SC_ATOMIC_DECLARE(uint64_t, drop);
@@ -32,14 +34,14 @@ typedef struct LiveDevice_ {
 } LiveDevice;
```

MIXED MODE : STEP 4

4. Figure out how to handle a packet

This is similar to point 3, but in this case we need to figure out what Suricata should do with a packet since it could come from different sources. It can be sent to IDS or IPS.

MIXED MODE : STEP 4

We add a mode to the packet structure which tells us if we have to IDS or IPS it :

```
diff --git a/src/decode.h b/src/decode.h
index f57dcea..3a008d2 100644
--- a/src/decode.h
+++ b/src/decode.h
@@ -351,6 +351,14 @@ typedef struct PktProfiling_ {
 /* forward declartion since Packet struct definition requires this */
 struct PacketQueue_;

+/*
+   since a pkt could come from different runmodes, we need to set
+   a mode to be able to do an action later.
+   (e.g. if a pkt come from NFQ, pkt_mode will be set to IPS and
+         it means that we'll be able to drop it.)
+*/
+enum PktMode {PKT_MODE_IDS, PKT_MODE_IPS, PKT_MODE_AUTO};
+
```

MIXED MODE : STEP 4

And the mode is set for each capture method source :

```
diff --git a/src/source-nflog.c b/src/source-nflog.c
index 7722244..a8d68b9 100644
--- a/src/source-nflog.c
+++ b/src/source-nflog.c
@@ -155,6 +155,7 @@ static int NFLOGCallback(struct nflog_g_handle *gh, struct nfgenmsg *msg,
         return -1;

     PKT_SET_SRC(p, PKT_SRC_WIRE);
+    p->pkt_mode = PKT_MODE_IDS;

     ph = nflog_get_msg_packet_hdr(nfa);
     if (ph != NULL) {
diff --git a/src/source-nfq.c b/src/source-nfq.c
index 38770cb..ce98984 100644
--- a/src/source-nfq.c
+++ b/src/source-nfq.c
@@ -505,6 +505,7 @@ static int NFQCallBack(struct nfq_q_handle *qh, struct nfgenmsg *nfmsg,
     }
     PKT_SET_SRC(p, PKT_SRC_WIRE);

+    p->pkt_mode = PKT_MODE_IPS;
     p->nfq_v.nfq_index = ntv->nfq_index;
     ret = NFQSetupPkt(p, qh, (void *)nfa);
     if (ret == -1) {
```

MIXED MODE : STEP 4

In this way we are able to drop a packet and log it :

```
diff --git a/src/output-json-alert.c b/src/output-json-alert.c
index 2c0d017..b3a1716 100644
--- a/src/output-json-alert.c
+++ b/src/output-json-alert.c
@@ -142,7 +142,7 @@ void AlertJsonHeader(const Packet *p, const PacketAlert *pa, json_t *js)
     char *action = "allowed";
     if (pa->action & (ACTION_REJECT|ACTION_REJECT_DST|ACTION_REJECT_BOTH)) {
         action = "blocked";
-    } else if ((pa->action & ACTION_DROP) && EngineModeIsIPS()) {
+    } else if ((pa->action & ACTION_DROP) && PacketModeIsIPS(p)) {
         action = "blocked";
     }

@@ -274,7 +274,7 @@ static int AlertJson(ThreadVars *tv, JsonAlertLogThread *aft, const Packet *p)

                 MemBufferReset(payload);

-                if (!EngineModeIsIPS()) {
+                if (!PacketModeIsIPS(p)) {
                     if (p->flowflags & FLOW_PKT_TOSERVER) {
                         flag = FLOW_PKT_TOCLIENT;
                     } else {
@@ -401,7 +401,7 @@ static int AlertJsonDecoderEvent(ThreadVars *tv, JsonAlertLogThread *aft, const
         char *action = "allowed";
         if (pa->action & (ACTION_REJECT|ACTION_REJECT_DST|ACTION_REJECT_BOTH)) {
             action = "blocked";
-        } else if ((pa->action & ACTION_DROP) && EngineModeIsIPS()) {
+        } else if ((pa->action & ACTION_DROP) && PacketModeIsIPS(p)) {
             action = "blocked";
         }
```

MIXED MODE : STEP 4

As you can see from the code, we check which the packet mode (PacketModeIsIPS()) is, instead of checking the engine mode (EngineModeIsIPS()).

DEMONSTRATION

Let's start with a demonstration now.

<real ninja mode>

Consider the following scenario :

We are using Suricata on a gateway that inspects all incoming traffic, and in particular we want to block all SSH connections from fake SSH agents.

DEMONSTRATION

Once Suricata detects an SSH connection, it outputs it to EVE (json logging).

Now, we should look at EVE file, take our suspiscious IP and add it to the set.

Is it right ?

DEMONSTRATION

NO ! :)

We are in real ninja mode now, we must find a better solution.

DEMONSTRATION

Solution :

Deny **O**n **M**onitoring

DOM is a tool, written by Eric Leblond, which implements a solution similar to fail2ban.

It parses the Suricata EVE log file searching for SSH events.

If the client version is suspicious, it adds the host to a blacklist by using ipset.

DEMONSTRATION

When DOM adds suspiscious IP to the set,

Suricata will IPS incoming connection from them.

DEMONSTRATION

Summary :

- Suricata inspects SSH incoming connection, and outputs it to the EVE log file
- DOM parses the EVE log file and, if necessary, adds a suspiscious IP to the set
- Suricata will IPS incoming connection from the IPs that are in the set

DEMONSTRATION

Let's implement the solution now.

At first, we need to create our set contaning all suspiscious IP :

```
nft add set filter suspisciousip { type ipv4_addr\;}
```

DEMONSTRATION

Then we add our ruleset to specify which traffic to inspect and block.

```
table filter {
    chain forward {
        ip saddr @suspectedip queue 0
        ip daddr @suspectedip queue 0
        log group 10 # send the rest of traffic through nflog
    }
}
```

DEMONSTRATION

In suricata configuration file, we must enable EVE logging with ssh :

```
# Extensible Event Format (nicknamed EVE) event log in JSON format
- eve-log:
    enabled: yes
    filetype: regular #regular|syslog|unix_dgram|unix_stream|redis
    filename: eve.json
    types:
      - ssh
```

DEMONSTRATION

We are ready to start suricata in mixed mode :

« suricata -c suricata.yaml -q 0 –nflog -v »

</real ninja mode>

CONCLUSION

- Mixed mode code is ready but isn't merged yet :
  https://github.com/glongo/suricata/tree/dev-mixed-mode-v4
- It still requires testing
- Feedback is appreciated

CONTACT

- Mail : glongo@stamus-networks.com
- Twitter : @theglongo
- IRC : glongo @ irc.freenode.org/#suricata

Q&A

Thank you for your attention !

Questions ?