Suricata IDPS and its interaction with Linux kernel

Eric Leblond, Giuseppe Longo
Stamus Networks
France, Italy
eleblond@stamus-networks.com glongo@stamus-networks.com

Abstract

Suricata is an open source network intrusion detection and prevention system. It analyzes the traffic content against a set of signatures to discover know attacks and also journalize protocol information.

One particularity of IDS systems is that they need to analyze the traffic as it is seen by the target. For example, the TCP streaming reconstruction has to be done the same way it is done on the target operating systems and for this reason it can't rely on its host operating system to do it.

Suricata interacts in a number of ways with the underlying operating system to capture network traffic. Under Linux it supports a wide range of capture methods ranging from AF_PACKET to NFQUEUE or NFLOG.

The purpose of this paper is to describe how some different performance challenges and interactions have been addressed with the Linux kernel and to show what works are in progress to increase performance. We will also explain in detail which are the current limitation, and some ideas that looked good at first, but wrong at the end. Finally, we will cover some possible evolutions like the offloading of some know good traffic.

Keywords

Suricata, Linux, Kernel, AF-PACKET, NFQUEUE, NFLOG

Introduction

Suricata [3] is an open source network intrusion detection and prevention system. It analyzes the traffic content against a set of signatures to discover know attacks and also journalize protocol information. A signature is a list of filters that apply to the traffic. For instance, Suricata can search a string in the content of a TCP stream or in the user agent seen in HTTP requests.

IDS specific constraints

IDS and evasion attacks

One particularity of IDS systems is that they need to analyze the traffic as it is seen by the target. If they don't do so then they may analyse a different traffic. This can lead to missed attacks (false negative) or to trigger alerts for benign traffic (false positive).

For example, the TCP streaming reconstruction has to be done the same way it is done on the target operating systems. All reconstructions that are handled differently by operating system due to RFC incompletion or bad implementation must be taken into account by the IDS. If streaming is not done this way, then attackers can abuse this interpretation mistake to get this attack unnoticed. For instance, if they attack a Windows operating system seen as a Linux by the IDS, they just need to use a TCP segmentation scheme that is done differently on Linux and on Windows so the IDS will see a content that is not the one received by the target. This kind of technique is called evasion [4] and it is available in attack tools just as Metasploit [2]. For this reason it can't rely on the operating system it is running on to do it.

Streaming in IDS and IPS mode

To be accurate, Suricata wants to analyze the traffic as it is seen by the host. In the case of TCP traffic, it is analyzing the data once it has been acked. This ensures that the host has received the data Suricata is going to analyse. This also permit to do the reconstruction based on the input of the target (like removing from the stream duplicate fragments that have not been acked).

But this approach does not work in Intrusion Prevention System (or IPS) mode because the data have reached the host that we protect before they get analyzed. And if, for example, we have an attack fitting into one single datagram then the targeted host has been compromised.

So another approach is necessary in IPS mode. When Suricata receives a packet, it triggers the reassembly process itself. Doing so it can analyze the data before they are sent to the target. With this approach, if the detection engine decides a drop is required, the packet containing the data itself can be dropped.

Doing the reassembly in Suricata has some consequences on the traffic. In some case, it may be necessary to modify packets. For example, in the case of overlapping segments some portion of the data in the packet may have to be overwritten.

So in IPS mode, Suricata is not only dropping packets but can also modify their content.

Suricata and performance

All signatures needs to be inspected at the difference of a firewall because the IDS needs to find if any signature is matching and multiple matches for a single packet are possible. Sig-

natures can be costly to evaluate as they can involve regular expression and/or Lua scripting. Current signatures sets contain between 15000 and 30000 signatures. Thus an iterative evaluation is not possible if the IDS needs to analyse traffic in real time. Some optimisations are needed to make high speed matching possible. Most used techniques is a combination of pre filtering by IP parameters and multi pattern matching on content filter (using mainly Aho-Corasick algorithm [1]).

Even with these optimisation the bandwidth per core is limited and a single core can not process more than 1 Gbps. Real life measurement give a limit of around 300 to 400 Mbps per core.

Running IDS on 10 Gbps network is frequent so IDSes needs to use some load balancing techniques to be able to scale.

Suricata and Linux kernel

Raw packet capture

Suricata principal capture method is AF_PACKET description. It uses the raw socket to capture a copy of the network traffic sent by a switch using port mirroring or by a tap device.

Suricata is using AF_PACKET fanout mode to do the load balancing needed to reach higher bandwidth. AF_PACKET fanout is doing a load balancing of one interface traffic over multiple sockets. Suricata is thus attaching multiple capture threads to the interfaces. Most common and fastest threads organization for Suricata AF_PACKET capture mode is shown in figure 2.

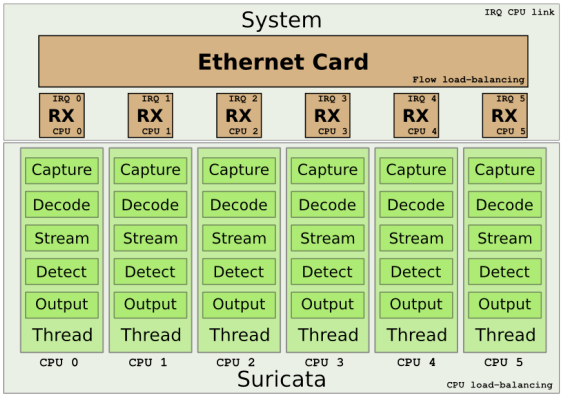

Figure 1: Suricata workers running mode

All capture threads runs all functional tasks in chains. Each thread is pinned to a core and system is tuned to attach the core to a RSS queue of the network card.

The load balancing has to be done with respect to the flow as spreading the packets for one flow other different sockets will result in the streaming engine seeing out of order packets for a single stream resulting in incapacity to get the stream information reconstruction correctly done.

So one problem of this running mode is that one single intensive flow is able to fill in the buffer on one receiver thread. Basically any flow which speed is superior to one single core treatment capacity will saturate the core and cause massive packet loss.

The rollover option has been added by Willem de Bruijn to AF_PACKET to address this issue. When the ring buffer of one socket is full the packets that should have given to that socket are given to the next socket with free space. Using this option in Suricata was straightforward and tests have been made. They have shown some interesting results.

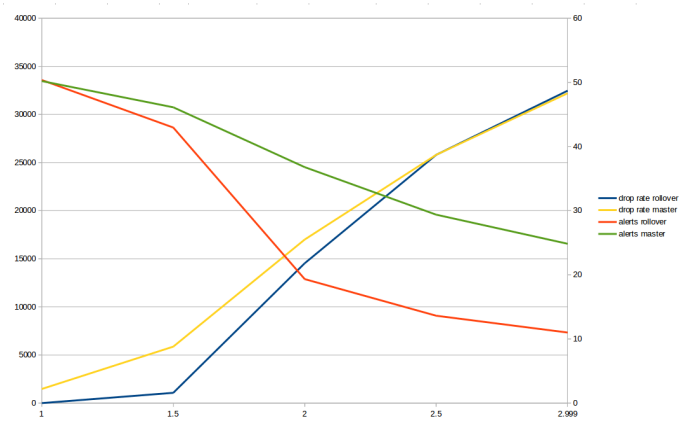

Figure 2: Accuracy test of Suricata using rollover mode (graph by Victor Julien)

If the drop rate is lower when using the rollover option, the number of alerts is lower when using rollover option. This means a important amount of traffic has not correctly being analyzed when rollover option has been used.

Explanation it the sending the traffic to another socket when one is full is useless for Suricata as it is causing massive out of order packets. This confused the streaming engine and result in an important number of flow to be non correctly analyzed.

Interaction with Netfilter

NFQUEUE In Intrusion Prevention mode, Suricata needs to be able to drop the packets if they contains something that is referenced as a malicious traffic. To do so, Suricata needs to use a different capture method.

The most used IPS capture method is Netfiter queue. It allows Suricata to perform action like DROP or ACCEPT on the packets.

The firewall ruleset needs to be updated. A new rule has to be added to send the traffic to the Suricata connected to a queue for a decision.

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

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

On performance side, 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.

NFLOG This is Netfilter logging via netlink socket. This capture method is similar to NFQUEUE but it only sends a copy of packet without issuing a verdict.

It's used to log packet by the kernel packet filter and pass it via userspace software. Suricata supports the NFLOG method to provide an IDS solution linked with Netfilter logging.

Mixed Mode

It's a capability of Suricata to capture packets from different sources and handle them differently depending on the source. For example, it's possible to setup Suricata as IPDS, as Prevention and Detection System, using NFQUEUE and NFLOG.

As seen in previously, the algorithm used for stream reassembly are different in IPS and IDS modes for security reason. So Suricata needs to be able to handle packets from NFQUEUE and NFLOG differently. So the key point of mixed mode is that Suricata decide on a per packet basis if it has to be inspected (IDS) or blocked (IPS).

One usage of mixed mode is the following scenario. Let's suppose we want manage a firewall protecting a public webserver. We would like to deploy some IDS/IPS ability to enhance the security of this server. The traffic on port 80 is too critical to be blocked so we would like to have only IDS on it. For the rest of the ports we can safely run IPS on them and block traffic if necessary. A classical setup would have required to run many Suricata instances with different configuration files. Instead in mixed mode, we can act as IDS to port 80 and block the rest of traffic which is less business sensitive. This means Suricata will act as IPS on other ports than 80.

To do so we just need to setup Suricata in mixed mode and have two different Netfilter filtering rules to send port 80 to NFLOG and send the rest to NFQUEUE.

Suricata and offloading

Stream reassembly depth

In most cases attack is done at start of TCP session. For some protocols generation of requests prior to attack is uneasy and is at least not common. For that reason, Suricata is reconstructing the stream till a certain configurable limit called stream reassembly depth. All packets of a flow after that limit are not sent to streaming engine. This result in lessen the load on the IDS. But this is not perfect as some work is still done inside Suricata for these packets.

Interest of offloading

The purpose of the Offloading API is to boost the performance during the packet acquisition, by ignoring some kind of intensive traffic. First objective is to completely ignore packets of a flow once stream reassembly depth is reached. Second objective is to disable almost instantly single intensive flow like Netflix. To do so we must ignore them as soon as we have identified them.

Implementation of framework

The API development consists in the implementation of a method that is called when Suricata decides to activate offloading.

Since Suricata works with several capture methods, the offloading method must be implemented in each of it, or at least, in the capture methods that we want to make able to offload traffic.

The offload can be applied in a general context, offloading all kind of traffic after that a TCP session is reconstructed enough, otherwise it's possible to offload a specific kind of traffic with the usage of signatures.

Use it with NFQ

The usage of offloading with NFQ, requires a Netfilter ruleset which will specify the packet's verdict.

```
table ip filter {
        chain forward {
                type filter hook forward priority 0;
                # usual ruleset
        }

        chain ips {
                type filter hook forward priority 10;
                meta mark set ct mark
                mark 0x00000001 accept
                queue num 0
        }

        chain connmark_save {
                type filter hook forward priority 20;
                ct mark set mark
        }
}
```

The purpose of the ruleset is to send all unmarked packets to nfqueue. This is done by using a dedicated chain coming after the firewall chain (hence we get all packets accepted by firewall policy). If they have a mark, we accept them, if not kernel is sending them to Suricata through nfqueue.

Inside Suricata, the offloading function is really simple. It consists in setting a dedicated offload mark to the packets:

```
static void NFQOffloadCallback(Packet *p)
{
    p->nfq_v.mark = (nfq_config.offload_mark
                     & nfq_config.offload_mask)
                    | (p->nfq_v.mark
                     & ~nfq_config.offload_mask);
    p->flags |= PKT_MARK_MODIFIED;

    return;
}
```

To activate the Offloading API, we can :

- Update Suricata configuration by enabling the stream.offloading setting to yes: in this case Suricata will trigger the callback
  when the stream.reassembly.depth setting is reached.
- Add the new 'offload' keyword in a signature, for example:

```
alert http any any -> any any (content:"netflix.com";
 http_host; offload; sid:1234; rev:1;)
```

When this rule is matched, Suricata will trigger the offloading function.

Tests have shown that using an intensive flow generated by iperf, we manage to trigger the offloading fast enough to almost get back to Netfilter raw performances.

Conclusion

Suricata has some constraints which are different than what is usually seen in the firewall and router world. Being CPU bound, it relies heavily on load balancing and due to current architecture, it can suffer from an intensive and long flow. Offloading API is a way to address that by shunting traffic not inspected by Suricata at the earliest stage. Current implementation of the offloading API is only done for NFQUEUE. We are planning to extend it to other capture methods such as AF_PACKET.

References

[1] Aho, A. V., and Corasick, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Commun. ACM* 18(6):333–340.

[2] Moore, H. D. Metasploit penetration testing software. http://www.metasploit.com.

[3] Open Information Security Foundation. Suricata. http://www.suricata-ids.org.

[4] Ptacek, T., and Newsham, T. 1998. Insertion, evasion, and denial of service: Eluding network intrusion detection. http://insecure.org/stf/secnet_ids/secnet_ids.html.