AUTOMATED NETWORK SECURITY WITH RUST: DETECTING AND BLOCKING PORT SCANNERS

Rédigé par Clément Fleury - 06/12/2024 - dans Développement

Did you ever wonder how IDS/IPS like Snort or Suricata were able to interact with the network stack of the Linux kernel ?

Do you also happen to like Rust ?

Well dear reader, this article is for you !

INTRODUCTION

In today's digital age, protecting networks from malicious activities is crucial. This project demonstrates how to build an automated system using Rust to detect and block suspicious network activity by leveraging Linux networking features.

While reading this article, keep in mind that this is a showcase of what could be done, but the applications are actually endless and there are probably greater things to do with the feature we are about to explore.

Objectives:

- Intercept packets at destination of closed ports.
- Track multiple failed connection attempts from a same IP address.
- Automatically block IPs that perform what looks like port scanning.

Even though it may sound like it, this article is **not** about how to properly block port scanners, it is about extending Linux's firewalling capabilities.

After a brief overview of the Netfilter framework, we will dive into the `nfnetlink_queue` feature which allows us to process packets in userspace.

WHAT IS NETFILTER ?

According to Wikipedia:

> Netfilter is a framework provided by the Linux kernel that allows various networking-related operations to be implemented in the form of customized handlers. Netfilter offers various functions and operations for packet filtering, network address translation, and port translation, which provide the functionality required for directing packets through a network and prohibiting packets from reaching sensitive locations within a network.

Netfilter is an essential component of the Linux kernel, offering robust mechanisms for network packet manipulation. By leveraging its hooks, chains, and rules, administrators can implement sophisticated firewall rules, perform NAT, manage traffic shaping, and enhance overall network security on Linux systems.

It forms the backbone of various network management tools, including `iptables`, `nftables`, and others.

In this article we will be focusing on the userspace tool `nftables`, which is the new cool kid on the block that aims to replace `iptables`, `ip6tables`, `arptables`, and `ebtables`.

TABLES

Netfilter organizes rules into chains and tables, allowing you to categorize various operations according to your needs.

Each table belongs to a specific protocol family and will only process packets that are part of the configured family.

You can find the `nft_table` struct in the kernel source code, here are some of the parts relevant to us as users:

```
struct nft_table {
    // [...]
    struct list_head chains;
    // [...]
    u16 family:6;
    // [...]
    char *name;
    // [...]
};
```

Here are the different supported protocol families you can operate on:

- **ipv4**: tables of this family can process IPv4 traffic
- **ipv6**: tables of this family can process IPv6 traffic
- **arp**: tables of this family can process ARP traffic
- **bridge**: table of this family can process traffic going through bridges
- **netdev**: this is a special family which allows the table to process **all** traffic going through a single network interface just after the NIC driver passes them up to the kernel network stack, packets have not been reassembled yet

(You can find them there in the kernel source code)

Here's how you would define a new table using `nftable` configuration file syntax :

```
table <family> <name> {
}
```

Note: even though there's no direct kernel support for it, the userspace tool `nftables` defines another table type called `inet` which allows you to operate on both IPv4 and IPv6 families at the same time.

TL;DR : tables are basically containers of chains that will apply to a specific protocol family.

CHAINS

Within each table, rules are organized into chains that define the sequence of processing.

There are three kinds of chains:

- **filter**: enables you to accept or drop packets
- **nat**: enables you to perform network address translation
- **route**: trigger a new route lookup in the kernel if the IP header has changed

Each chain is given a hook name to use, it allows you to choose at which stage of the kernel network stack your rules should apply.

Netfilter is composed of 5 main hooks which are specific points in the kernel's networking stack where packets can be intercepted. You can find in the kernel source code the enum representing those hooks:

```
enum nf_inet_hooks {
    NF_INET_PRE_ROUTING,
    NF_INET_LOCAL_IN,
    NF_INET_FORWARD,
    NF_INET_LOCAL_OUT,
    NF_INET_POST_ROUTING,
    NF_INET_NUMHOOKS,
    NF_INET_INGRESS = NF_INET_NUMHOOKS,
};
```

1. **NF_INET_PRE_ROUTING**: Called when a packet arrives at the network interface but before routing decisions.
2. **NF_INET_LOCAL_IN**: Called for packets destined to be delivered locally after routing.
3. **NF_INET_FORWARD**: Called for packets that are being forwarded through the host (i.e. not destined for local delivery).
4. **NF_INET_LOCAL_OUT**: Called for packets originating from the host before they enter the network interface.
5. **NF_INET_POST_ROUTING**: Called after routing decisions but just before packets leave the network

Each of these hooks map to the following `nftables` names (order has been preserved) :

1. **prerouting**
2. **input**
3. **forward**
4. **output**
5. **postrouting**

Netfilter also has a priority mechanism where lower numbers indicate higher priority. This ordering allows different components to process packets in a specific sequence, enabling complex filtering, logging, and NAT operations while ensuring that critical tasks are performed before less urgent ones. By default the kernel sets a number of hooks with a given priority, it allows you to perform some operations before or after some of the default ones.

For example here are some of the default priorities defined in the kernel source code:

- **NF_IP_PRI_RAW_BEFORE_DEFRAG** (-450): hooks that have this priority will be called before any packet defragmentation has been done (this is where we will operate later on)
- **NF_IP_PRI_CONNTRACK_DEFRAG** (-400): this is where packets defragmentation happens
- **NF_IP_PRI_CONNTRACK** (-200): this is where conntrack kicks in and associate packets with tracked connections
- **NF_IP_PRI_NAT_DST** (-100): this is where DNAT happens
- **NF_IP_PRI_FILTER** (0): this is where packet filtering is performed
- **NF_IP_PRI_NAT_SRC** (100): this is where SNAT happens

Here's how you would define a new chain using `nftables` configuration file syntax :

```
table inet my_table {
    chain my_chain {
        type <chain_type> hook <hook_type> priority <priority>;
    }
}
```

TL;DR : chains are basically containers of rules that will apply at a specific stage of the network stack.

RULES

Within each chain, you can define a set of rules that packets will be matched against.

Considering the amount of rules available, we won't get into the specifics of all of them, you can refer to the nftables wiki for that.

Rules are processed sequentially. Note that if one of the rules triggers the acceptance or denial of a packet, the other rules that follow won't be processed.

A rule is composed of:

- Packet matching criteria (optional)
- An action to perform

If no criteria are defined, any packet that make it to that rule will trigger it, for example:

```
# Conditional rule, triggers a packet acceptance if it is a TCP packet intended for port 22
tcp dport 22 accept

# Unconditional rule, triggers logging of all the packets that made it to this rule
log flags all prefix "THIS PACKET IS NOT INTENDED FOR PORT 22: "
```

Here are some of the actions that can be triggered by a rule:

- Drop (ignore) packets
- Reject packets (inform the other end of the communication)
- Accept packets
- Count packets
- Log packets
- **Send a packet to userspace for further analysis**

As you might have guessed, the chapters that follow will focus on the last item from this list :)

SEND PACKETS TO USERSPACE

The Linux kernel has a feature called the `nfnetlink_queue` which enables us to send packets to a userspace program for further processing. For that program to be able to handle the packets queue, it must either run as root or have the `CAP_NET_ADMIN` capability. The userspace program has the ability to decide whether the packet should be accepted or not, it is called a verdict in Netfilter jargon.

Here's what you can find in `libnetfilter_queue` documentation about those verdicts:

- NF_DROP Drop the packet. This is final.
- NF_ACCEPT Accept the packet. Processing of the current base chain and any called chains terminates, but the packet may still be processed by subsequently invoked base chains.
- NF_STOP Like NF_ACCEPT, but skip any further base chains using the current hook.
- NF_REPEAT Like NF_ACCEPT, but re-queue this packet to the current base chain. One way to prevent a re-queueing loop is to also set a packet mark using `nfq_nlmsg_verdict_put_mark()` and have the program test for this mark; or have the nefilter rules do this test.
- NF_QUEUE_NR(new_queue) Like NF_ACCEPT, but queue this packet to queue number new_queue. As with the command-line queue num verdict, if no process is listening to that queue then the packet is discarded

Important: once a packet is sent to userspace for processing, there's no way to continue processing the current `nftables` chain the packet is at, the userspace program will be the last one to issue a verdict for that chain.

The `nfnetlink_queue` has two modes of operation :

- Copy metadata only
- Copy metadata and payload content (up to 64KiB)

The latter is obviously more resource intensive.

Note that the IP header won't be in the metadata, here's a non-exhaustive list of what you can find in it:

- packet length
- network interface the packet arrived on
- network interface the packet is transmitted from

- MAC address
- uid/gid of the process sending/receiving the packet

However some of these metadata won't be available depending on the stage at which you send the packet to the queue. For example if you send the packet to userspace in a postrouting chain, you won't be able to get the network interface the packet was coming from.

And now comes the magic `nftables` statement that will instruct the kernel to put packets in the queue for userspace processing:

```
queue num 0
```

The value `0` is the ID of the queue, the userspace program has to explicitly bind to that ID.

Obviously you can still filter the packets that will go the queue, for example the following will only send TCP packets at destination of port 1337 to the queue:

```
tcp dport 1337 queue num 0
```

Additionally, it must be noted that `nftables` allows you to create multiple queues in a single statement, using the following syntax:

```
queue num 0-3
```

It can be really useful if you want to be able to perform load balancing across multiple threads in your userland program.

There are two additional flags available to you:

- `bypass` : tell the kernel to **accept** the packet if no userspace program binds to that queue, which might have serious security consequences if not done correctly.
- `fanout` : an alternative algorithm for deciding to which queue each packet should go.

The `fanout` option is unfortunately not very well documented, so bare with me.

The kernel implements various queue dispatching algorithms:

```
switch (f->type) {
case PACKET_FANOUT_HASH:
default:
  idx = fanout_demux_hash(f, skb, num);
  break;
case PACKET_FANOUT_LB:
  idx = fanout_demux_lb(f, skb, num);
  break;
case PACKET_FANOUT_CPU:
  idx = fanout_demux_cpu(f, skb, num);
  break;
case PACKET_FANOUT_RND:
  idx = fanout_demux_rnd(f, skb, num);
  break;
case PACKET_FANOUT_QM:
  idx = fanout_demux_qm(f, skb, num);
  break;
case PACKET_FANOUT_ROLLOVER:
  idx = fanout_demux_rollover(f, skb, 0, false, num);
  break;
case PACKET_FANOUT_CBPF:
case PACKET_FANOUT_EBPF:
  idx = fanout_demux_bpf(f, skb, num);
  break;
}
```

By default (i.e. without the `fanout` flag), a hash of each incoming packet metadata will be performed to decide which queue it should go to. Here are the metadata the kernel will hash to decide where to send the packet:

```
struct flow_keys {
    struct flow_dissector_key_control control;
#define FLOW_KEYS_HASH_START_FIELD basic
    struct flow_dissector_key_basic basic __aligned(SIPHASH_ALIGNMENT);
    struct flow_dissector_key_tags tags;
    struct flow_dissector_key_vlan vlan;
    struct flow_dissector_key_vlan cvlan;
    struct flow_dissector_key_keyid keyid;
    struct flow_dissector_key_ports ports;
    struct flow_dissector_key_icmp icmp;
    /* 'addrs' must be the last member */
    struct flow_dissector_key_addrs addrs;
};
```

It basically allows your userspace program to receive on the same queue packets that are correlated to each others. If you use the `fanout` flag however, the current CPU ID will be used, and no hash will be performed:

```
static unsigned int fanout_demux_cpu(struct packet_fanout *f,
                     struct sk_buff *skb,
                     unsigned int num)
{
    return smp_processor_id() % num;
}
```

As of today only the metadata hash dispatching and the CPUID dispatching are supported by nftables on the userland side:

```
#define NFT_QUEUE_FLAG_BYPASS       0x01 /* for compatibility with v2 */
#define NFT_QUEUE_FLAG_CPU_FANOUT   0x02 /* use current CPU (no hashing) */
#define NFT_QUEUE_FLAG_MASK     0x03
```

- If you want raw performance, in most contexts you should pick the CPUID dispatcher (i.e. `fanout` flag).
- If you want to receive packets that correlate to each others on the same queue, use the default dispatcher (i.e. **no** `fanout` flag).

WRITING THE PROGRAM

For demonstration purposes, we will write a program that automatically bans IP addresses that try to reach multiple closed ports in a row, which most likely indicates that someone is scanning the ports of our machine.

Considering that our code will be network facing, it makes a lot of sense to write it in a high level language, to avoid any memory handling pitfalls that could lead to a vulnerability. This is why we will write our program using the Rust programming language, which is perfect for this job.

We will use the Rust crate called nfq which is a pure Rust implementation that does not rely on the `libnetfilter_queue` C library. On other hand, it is less featureful and might prevent us from using interesting `nfnetlink_queue` APIs.

Before we dive into any programming, we set up an Ubuntu VM with the following minimal `nftables` configuration :

```
#!/sbin/nft -f

flush ruleset

table inet synacktiv_table {
  chain synacktiv_chain {
    type filter hook input priority 0; policy drop;
    ct state invalid counter drop comment "early drop of invalid packets"
```

```
    iif lo accept comment "accept loopback"
    iif != lo ip daddr 127.0.0.1/8 counter drop comment "drop connections to loopback not coming from loop
back"
    iif != lo ip6 daddr ::1/128 counter drop comment "drop connections to loopback not coming from loopbac
k"

    # Accept already established/related connections
    ct state {established, related} counter accept comment "accept all connections related to connections
made by us"

    # Setup services
    tcp dport 22 accept comment "accept all ssh connections"
    tcp dport 80 accept comment "accept web server connections"

    # Send remaining packets to userspace
    queue num 0
  }
}
```

Notice how the `queue` statement is placed at the end of the `synacktiv_chain`, any packet that has been accepted already by the precious rules won't be sent to userspace.

Using nfq to receive packets is very straightforward:

```
use nfq::{Queue, Verdict};

let mut queue = Queue::open()?;
queue.bind(0)?;

loop {
    let mut msg = queue.recv()?;
    let packet = msg.get_payload();

    if let Err(err) = handle_packet(packet) {
        log::warn!("Unable to process packet: {err}");
    }

    msg.set_verdict(Verdict::Drop);
    queue.verdict(msg)?;
}
```

In the above snippet we:

- Bind to the queue ID 0
- Retrieve the payload of the `nfnetlink_queue` message (the packet itself from which we will parse the IP header)
- Set the verdict to `Drop` no matter what: packets that were sent to userspace didn't match any rule and were therefore supposed to be dropped

Remember that our goal is to ban IP addresses. In order to access those, we must first parse the IP header. For that purpose we will use the pnet crate which provides ways to parse many kinds of packets from different layers. For now our `handle_packet()` function will only log the IP address and the destination port this IP tried to reach:

```
use pnet::packet::{
    ipv4::Ipv4Packet,
    ipv6::Ipv6Packet,
};

fn handle_packet(packet: &[u8]) -> anyhow::Result<()> {
    let (addr, port) = if let Some(ipv4_packet) = Ipv4Packet::new(packet) {
        get_ipv4_source_and_dest_port(ipv4_packet)?
    } else if let Some(ipv6_packet) = Ipv6Packet::new(packet) {
        get_ipv6_source_and_dest_port(ipv6_packet)?
```

```
    } else {
        bail!("packet does not belong to the IP layer");
    };

    log::info!("{addr} tried to reach port {port}");
    Ok(())
}
```

As you can see, parsing an IP packet using `pnet` is as simple as passing the raw packet bytes (`packet: &[u8]`) to `Ipv4Packet::new()` or `Ipv6Packet::new()`. These functions will return `None` if the packet does not have a valid IP header. Getting the IP address is as simple as calling the `get_source()` function on `Ipv4Packet` or `Ipv6Packet`.

Now that we have the source address, we need to parse the TCP/UDP header. Again, this is very easy thanks to `pnet`:

```
use std::net::IpAddr;
type Dport = u16;

fn get_ipv4_source_and_dest_port(packet: Ipv4Packet) -> anyhow::Result<(IpAddr, DPort)> {
    let port = match get_dest_port(packet.get_next_level_protocol(), packet.payload()) {
        Ok(port) => port,
        Err(err) => {
            bail!("unable to get IPv4 destination port ({err})");
        }
    };

    Ok((packet.get_source().into(), port))
}

fn get_dest_port(proto: IpNextHeaderProtocol, payload: &[u8]) -> anyhow::Result<DPort> {
    match proto {
        IpNextHeaderProtocols::Tcp => match TcpPacket::new(payload) {
            Some(tcp_packet) => Ok(tcp_packet.get_destination()),
            None => Err(anyhow!("invalid TCP packet")),
        },
        IpNextHeaderProtocols::Udp => match UdpPacket::new(payload) {
            Some(udp_packet) => Ok(udp_packet.get_destination()),
            None => Err(anyhow!("invalid UDP packet")),
        },
        other => Err(anyhow!("unsupported protocol: {other}")),
    }
}
```

At this point you might be wondering why we retrieve the destination port from the TCP/UDP layer. The reason is quite simple: in order to avoid unfair banning of IP addresses, we will only ban the ones that try to connect to multiple different ports. Some applications like `curl`, retry requests if they don't succeed, resulting in multiple connections to a potentially closed port. If we decided to ban IP addresses regardless of the ports they tried to reach, a single `curl` command on a closed port could result in a ban.

Now that we are able to get both the IP address and the destination port, we need to implement the banning mechanism.

We will start by creating a struct that will hold the IP address of a remote client as well as all the closed ports it has tried to reach:

```
struct SuspiciousClient {
    addr: IpAddr,
    reached_ports: HashSet<DPort>,
}
```

The `HashSet` is here to ensure that we only have a list of unique ports. If a client tries to reach the same port multiple times, it will only count as one closed port attempt.

For the sake of this article's length I will not show all the `SuspiciousClient` logic but the idea is that as soon as a client reaches the maximum predefined amount of different ports, it will get banned:

```
const ALLOWED_PORTS_COUNT_BEFORE_BAN: usize = 10;

if client.reached_ports.len() >= ALLOWED_PORTS_COUNT_BEFORE_BAN {
    match ban_suspicious_client(client) {
        Ok(()) => {
            log::info!("{client_addr} has been banned");
        }
        Err(err) => {
            // If the ban didn't succeed, we keep the suspicious client in the vec for later retry
            log::error!("Failed to ban client: {err}");
        }
    }
}
```

In order to ban a client, we must add a new firewall rule, there are two main ways of doing so:

- Using the low level `libnftnl` library which provides a Netlink API to the in-kernel `nf_tables` subsystem
- Using the `libnftables` library which provides a high level JSON API as an alternative frontend to the `nft` CLI

The only Rust crate that comes close to the second option is `nftables`. Unfortunately, this crate is a wrapper around the `nft` CLI tool and not an actual alternative frontend like `libnftables`. We potentially have to process a lot of packets, we cannot afford the cost of starting an external executable.

Therefore, we decided to use the `rustables` crate instead, which is a low level wrapper around `libnftnl`.

Because of it's low level nature, we have to handle many things ourselves, here's what their developers say about it:

> This library currently has quite rough edges and does not make adding and removing netfilter entries super easy and elegant. That is partly because the library needs more work, but also partly because nftables is super low level and extremely customizable, making it hard, and probably wrong, to try and create a too simple/limited wrapper.

The `rustables` crate works by sending batches of Netlink messages to the kernel. A batch is a bit like an SQL transaction, it contains a list of statements that must be executed atomically by the kernel. Here's how you can create a batch that will create a table and an associated chain:

```
use rustables::{Batch, Chain, MsgType, ProtocolFamily, Table};

const NETFILTER_HOOK_PRIORITY: i32 = libc::NF_IP_PRI_RAW_BEFORE_DEFRAG;

// Create a new batch.
let mut batch = Batch::new();

// Create a new table.
let table = Table::new(ProtocolFamily::Inet).with_name(NETFILTER_TABLE_NAME);

// Create a new chain.
let chain = Chain::new(&table)
    .with_name(NETFILTER_CHAIN_NAME)
    .with_hook(Hook::new(HookClass::PreRouting, NETFILTER_HOOK_PRIORITY));

// Add both the table and the chain to the batch.
batch.add(&table, MsgType::Add);
batch.add(&chain, MsgType::Add);

// Send the batch to the kernel.
batch.send().unwrap();
```

Notice how we used the `NF_IP_PRI_RAW_BEFORE_DEFRAG` priority for the prerouting hook, that way packets sent to our machine by the IP addresses that got banned will be dropped very early in the kernel, before any packet defragmentation or connection tracking happens. It will spare you some precious CPU cycles you don't want to spend for a potential attacker :)

The above snippet was for illustration proposes but cannot be used as such, there are a few things to consider.

First, we must make sure our program is idempotent: any attempt to create a table or a chain that already exists will result in an error, we must reuse the existing table/chain if they were created already by previous runs of our code.

We must also consider that the tables/chains/rules can change at any time: any attempt to create a chain to a table that doesn't exist anymore will result in an error, as always proper error handling is a must have.

Here's how one could write idempotent code that will either reuse or create a table:

```
// Retrieve existing tables.
let mut tables = rustables::list_tables()?;

// Look for our own table.
let existing_table_index = tables.iter().position(|t| {
    t.get_name()
        .is_some_and(|name| name == NETFILTER_TABLE_NAME)
});

let table = match existing_table_index {
    // The table exists already, reuse it.
    // By removing the value from the vec, we take ownership of it.
    Some(index) => tables.remove(index),

    // The table doesn't exist yet, add it to the batch.
    None => Table::new(ProtocolFamily::Inet)
        .with_name(NETFILTER_TABLE_NAME)
        .add_to_batch(&mut batch),
};

let mut chains = rustables::list_chains_for_table(&table)?;
// Do the same for our NETFILTER_CHAIN_NAME chain.
```

Now that we have everything we need to ban an IP address, we can finally finish the implementation of our `ban_suspicious_client()` function. Here's the final missing bit, a rule that will drop any packet which matches the IP address of the given client:

```
use rustables::Rule;

Rule::new(&chain)?
    .match_ip(suspicious_client.addr, /* source: */ true)
    .drop()
    .add_to_batch(&mut batch);

batch.send()?;
```

Let's put everything together, compile the project, run it inside our Ubuntu VM, and start port scanning using `nmap`.

```
$ sudo ./nmap-farewell
INFO  nmap-farewell > Listening to queue 0
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {36288} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {36288, 30963} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {36288, 30963, 43562} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {30963, 18419, 36288, 43562}
}
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {30963, 18419, 36288, 43562,
62195} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {31157, 30963, 18419, 36288,
```

```
43562, 62195} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {31157, 30963, 18419, 53464,
36288, 43562, 62195} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {62195, 30963, 53464, 18419,
30685, 36288, 31157, 43562} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {62195, 30963, 53464, 13337,
18419, 30685, 36288, 31157, 43562} }
DEBUG nmap-farewell > SuspiciousClient { addr: 192.168.122.1, reached_ports: {62195, 30963, 53464, 13337,
18419, 30685, 41829, 36288, 31157, 43562} }
INFO  nmap-farewell > 192.168.122.1 has been banned
```

```
$ sudo nft list ruleset
table inet synacktiv_table {
    chain synacktiv_chain {
        type filter hook input priority filter; policy drop;
        ct state invalid counter packets 0 bytes 0 drop comment "early drop of invalid packets"
        iif "lo" accept comment "accept loopback"
        iif != "lo" ip daddr 127.0.0.0/8 counter packets 0 bytes 0 drop comment "drop connections to loopb
ack not coming from loopback"
        iif != "lo" ip6 daddr ::1 counter packets 0 bytes 0 drop comment "drop connections to loopback not
coming from loopback"
        ct state { established, related } counter packets 567 bytes 4725518 accept comment "accept all con
nections related to connections made by us"
        tcp dport 22 accept comment "accept all ssh connections"
        tcp dport 80 accept comment "accept web server connections"
        queue to 0
    }
}
table inet port_scanners_auto_ban {
    chain prerouting {
        type filter hook prerouting priority -450; policy accept;
        ip saddr 192.168.122.1 drop
    }
}
```

It worked flawlessly ! 🥳

A few other things not covered in this article were implemented, most notably:

- Remove all the existing bans on startup, for reentrancy purposes
- Keep track of banned clients and automatically remove their IP from the ban list after a given period of time, this is helpful in case:
    - You accidentally lock yourself out of a remote server
    - You ban a public IP address that will later be assigned to someone else
    - You ban a private IP address that would be a different machine on another LAN (and you suspended your laptop for example)
- Various performance optimizations including:
    - Issue a drop verdict as soon as we receive a packet, so that the kernel doesn't wait for us
    - Start multiple threads to allow load balancing across multiple queues

CONCLUSION

In this article we dived into some of the Netfilter inner working and also explored its niche queuing feature. Thanks to it we were able to demonstrate how easy it is to extend the firewalling capabilities of `nftables`.

This project also demonstrates the power of combining Rust's system programming capabilities with Linux's networking features to create robust network security solutions. By following this example, you can build upon these concepts to develop more sophisticated and specialized tools for your network security needs.

After I finished writing this article and submitted it for review, someone at Synacktiv told me that nftables is already able to do what we've been doing in this article. If you are interested in implementing such banning mechanisms, you should definitely use that instead. My hope is that some people will still find this article to be a valuable resource if they want to write their own nftables "extension".

Feel free to explore the full source code here on GitHub.

I hope you enjoyed this article, until next time...