# Network Security
# Group 8 - Project Report

Alessio Campanelli (878170) Elia Bertapelle (881359)
Mario Gottardo (879088) Simone Jovon (882028)

A.Y. 2022/2023

**Abstract**

The goal of the project is to implement an IDS/firewall in the Linux Operative System, using a kernel process to intercept and filter the traffic, and an user space daemon that runs an analysis on the traffic using machine learning techniques.
GitHub repository URL: https://github.com/Gotti27/2columnS

# Contents



# Group and Activity Description

| Group member | Work done |
|---|---|
| Alessio Campanelli | Kernel module and Firewall |
| Elia Bertapelle | Research, Socket Communication and Debugging |
| Mario Gottardo | User Space Daemon, Traffic Classification ML model and Firewall |
| Simone Jovon | Socket Communication and User Space Daemon |

# 1 Introduction

The goal of the project was to develop a Network Intrusion Detection System that logs on the system journal in case of a network intrusion. Moreover, the system allows to filter the incoming traffic following some simple custom policies, set using a JSON file.

In modern network systems, NIDS and firewalls are essential to protect them and other assets. Applying Machine Learning techniques could represent the next stage of these tools.

The tool we have developed is composed of a kernel module which relies on `netfilter` and a user space daemon for traffic flow classification and user interaction.

# 2 Kernel Module

The first thing we need to do is create a program that intercepts the traffic, inspects it and then sends it to the user space daemon for the analysis. Since packet processing is done within the kernel, we must implement a kernel module that does all the above. Modern Linux allows us to use *Loadable Kernel Modules* in order to extend the kernel, without having to modify directly the base kernel.

We used the `netfilter` framework to facilitate the manipulation of packets. It works by inserting some hooks in various places, in our case in the INPUT chain, and connecting our program to said hook.

Firstly, we had to create an initialization (`firewall_init`) and a cleanup (`firewall_exit`) function. Then, after declaring the hook in a static variable, we filled its struct with the required information to let it operate, inside the initialization function. At this point, our code looks like this:

```
1  static struct nf_hook_ops hook_in;
2
3  unsigned int firewall_main(void* priv, struct sk_buff* skb, const struct
       nf_hook_state *state){
4      // ...
5  }
6
7  int firewall_init(void){
8      printk(KERN_INFO "-- Registering Filters --\n");
9      hook_in.hook = firewall_main;       // Hook's function
10     hook_in.hooknum = NF_INET_LOCAL_IN; // Hook type: INPUT chain
11     hook_in.pf = PF_INET;               // IPv4 Internet protocol family
12     hook_in.priority = NF_IP_PRI_FIRST; // Highest priority
13     nf_register_net_hook(&init_net, &hook_in);
14
15     // ...
16 }
17
18 void firewall_exit(void){
19     printk(KERN_INFO "-- Removing Filters --\n");
20     nf_unregister_net_hook(&init_net, &hook_in);
21
22     // ...
23 }
24
25 module_init(firewall_init);
```

```
26 module_exit(firewall_exit);
27
28 MODULE_LICENSE("GPL");
```

This is everything we need to make the kernel module work and let us read, filter and eventually modify all the packets arriving to our machine. Now we will focus on the `firewall_main` function, and how we read the data contained in the packets.

Since we are working on a low level of the OS, we are able to see every bit of the packet. This means that we have to split the different headers in different variables (in Figure 1 we are looking at the bottom row). This is an easy task since `netfilter` already gives us the functions to extract a pointer to the start of each header.

```
1     struct iphdr *iph;
2     struct tcphdr *tcph;
3     struct ethhdr *ether;
4
5     ether = eth_hdr(skb);
6     iph = ip_hdr(skb);
7     tcph = tcp_hdr(skb);
```

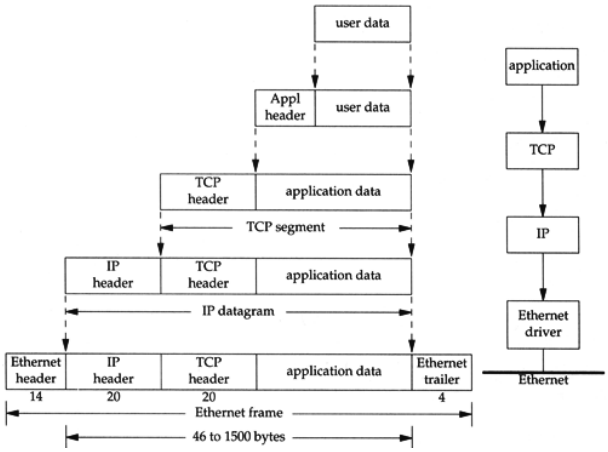


**Figure 1:** Packet encapsulation

Then from the different header variables we can read all the information stored inside each packet.

For now we are not interested into taking filtering decisions based on the packet content, so we are simply going to accept everything by ending the function with

```
1     return NF_ACCEPT;
```

## 2.1 Compiling and running the module

This is the makefile we used for compiling the kernel module:

```
1 obj-m += firewall.o
2
3 all:
4   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

This code was directly taken from "SEED labs - Firewall Exploration Lab"[Du]. We can clearly see that this is different from a typical user space program makefile. It also produces different files, of which we are mainly interested in `firewall.ko`, which is the compiled kernel module.

Then, to load it into the kernel, we just have to run

```
$ sudo insmod firewall.ko
```

and to read the logs

```
$ dmesg -Hw
```

# 3 Socket Communication

Once we intercept the packets from the kernel module, we need to send them to the user space daemon that will analyze the traffic. To do so, we decided to use Netlink sockets.

Netlink is used to transfer information between the kernel and user-space processes. It consists of a standard socket-based interface for user space processes and an internal kernel API for kernel modules[1]. Netlink sockets use the address family AF_NETLINK, as compared to AF_INET used by a TCP/IP socket.

We used Netlink because, unlike other communication mechanisms, it offers advanced communication features and has the following advantages over other them:

- Is asynchronous as it provides queues, not interfering with kernel scheduling.
- Provides a full-duplex communication link
- Provides the possibility of multicast communication
- Has less overhead compared to standard UDP sockets

## 3.1 Netlink implementation - Kernel

The first thing we have to do to in order to receive message from user space is define a hook to a method that will be called every time the socket receive a new message.
Then, we setup the socket with `netlink_kerne_create` passing: network namespace, netlink protocol, and a reference to optional parameters (like our hook).

```
int firewall_init(void){
    struct netlink_kernel_cfg cfg = {
        .input = netlink_handler,
    };

    // ...

    printk(KERN_INFO "-- Initializing Netlink --\n");
    nl_sk = netlink_kernel_create(&init_net, NETLINK_USERSOCK, &cfg);
    if (!nl_sk) {
        printk(KERN_CRIT "Error creating socket.\n");
        return -10;
    }
```

[1] man Linux

```
14
15      return 0;
16  }
```

This is the method that will be invoked every time a new packet is incoming and here we are going to perform the message sending to the user space.

```
unsigned int firewall_main(void* priv, struct sk_buff* skb, const struct
    nf_hook_state *state){
    struct nlmsghdr *nlh;
    struct sk_buff *skb_out;
    int msg_size, res;
    unsigned int target;

    // ...
```

Here we are going to check if the pid is initialized. If not, we are going to directly apply the rules that has been already configured. Note that the pid is a port ID and there isn't necessarily a 1:1 binding with the process ID.

```
    if (pid == -1) {
        printk(KERN_ERR "Pid or Buffer not configured\n");
        return convert_target(target);
    }
```

Now we initialize a new message buffer with a max message size of 100 bytes. If an error occurs during this process, an error message will be printed and the already configured rules will be applied.

```
    msg_size = 100;
    skb_out = nlmsg_new(msg_size, GFP_KERNEL);
    if (!skb_out) {
        printk(KERN_ERR "Failed to allocate new skb\n");
        return convert_target(target);
    }
```

The next step is the generation of the message: after the generation of a pointer of the message, we setup a new string with all the features that we need in order to correct analyze the incoming traffic searching for anomalies.

```
    nlh = nlmsg_put(skb_out, 0, seq++, NLMSG_DONE, msg_size, 0);
    snprintf(nlmsg_data(nlh), msg_size, "%pI4,%pI4,%d,%d,%x:%x:%x:%x:%x
    :%x,%x:%x:%x:%x:%x:%x,%02x,%hu,%c,%c,%llu",
             &(iph->saddr), &(iph->daddr), // src/dest ip
             ntohs(tcph->source), ntohs(tcph->dest), // src/dest port
             ether->h_source[0], /*...*/, ether->h_source[5], // src MAC
             ether->h_dest[0], /*...*/, ether->h_dest[5], // dest MAC
             iph->protocol, // ip protocol
             ntohs(iph->tot_len), // total packet length
             tcph->syn ? '1' : '0', // syn flag
             tcph->ack ? '1' : '0', // ack flag
             ktime_get_real_ns()
        );
```

Then, the last step, we try to send the message to the user space. If sending fails it means that no one is listening and so we are going to reset the pid to -1.

```
    res = nlmsg_multicast(nl_sk, skb_out, 0, NETLINK_GROUP, GFP_KERNEL);
    if (res < 0){
```

```
        printk(KERN_ERR "Error while sending to user\n");
        pid = -1;
    }

    // ...
}
```

## 3.2 Netlink implementation - User Space

On User Space side we need to create a netlink socket listening for message of the NETLINK_MYGROUP multicast group. Then, we send a netlink message to the Kernel Module to communicate that the program is listening on the socket.

```
# ...

NETLINK_MYGROUP = 2

sock = socket.socket(socket.AF_NETLINK, socket.SOCK_RAW, socket.
    NETLINK_USERSOCK)
sock.bind((0, NETLINK_MYGROUP))

log.info("Sending msg")
msg_data = b"Netlink connection established\x00"
msg_len = 16 + len(msg_data)
msg = msg_len.to_bytes(4, 'little') + b"\x03\x00" + b"\x00"*2 + b"\x00"
    *4 + b"\xb3\x15\x00\x00" + b"Netlink connection established\x00"

sock.send(msg)

# ...
```

With the same principle, we also created an API that manages to send netlink messages to the Kernel Module, containing the firewall rules to be enforced. The API takes as input a dictionary wrapping the rule to be sent, that is then converted into a byte string (which is the memory representation of the struct in C) and finally sent on the socket.

```
# ...

log.info("sending rules to kernel module")

def send_msg_to_kernel(msg, rule: bool):
    rule_data = b''
    if rule:
        source = msg['source'].encode().ljust(16, b'\x00')
        destination = msg['destination'].encode().ljust(16, b'\x00')
        port = msg['port'].to_bytes(2, 'little')
        protocol = msg['protocol'].to_bytes(1, 'little')
        action = msg['action'].to_bytes(1, 'little')

        rule_data = b'RULE:1'+source+destination+port+protocol+action
        log.info(rule_data)
    else:
        rule_data = f"RULE:0{msg}".encode()

    rule_len = 16 + len(rule_data)
```

```
    rule_msg = rule_len.to_bytes(4, 'little') + b"\x03\x00" + b"\x00"*2
    + b"\x00"*4 + b"\xb3\x15\x00\x00" + rule_data

    sock.send(rule_msg)

# ...
```

To finish, the following code reads netlink messages from the socket that are forwarded from the Kernel Module and retrieves the message payload containing the packets data.

```
log.info("Starting reciving.....")

# ...

while True:
    data = ''
    try:
        data = sock.recvmsg(1024)[0]
    except OSError:
        # ...
        continue

    msg_hdr = data[:16]

    msg_len,msg_type,flags,seq,pid = struct.unpack("=LHHLL", msg_hdr)
    msg_data = data[16:msg_len].strip(b'\x00').decode()

    p = msg_data.split(',')

    # ...
```

# 4 User Space Daemon

The goal of the user space daemon is to invoke our classifier and notify the user if the incoming packet flow looks malicious.

To integrate it as a background user service, the user space daemon will be written in Python3 and will be executed as a Linux service using Systemd.

## 4.1 Traffic classification and anomaly detection

Given the host incoming traffic, the user space has to analyze it and detect if an attack is ongoing. The problem can be addressed as detecting traffic anomalies.

Since our modules are operating at a pre-application level, they should not be able to understand the content of encrypted payloads like the ones protected by TLS/SSL. This implies that we can only rely on the packet headers.

The classifier has to be fast and easy to train, so the best choice is a Support Vector Machine (SVM), in particular a One-Class SVM. The kernel of this type of SVMs is circular, so they highlight a circular area in the hyper-dimensional feature space, splitting it in two regions: the inner one for non-anomalies and the outer one for anomalies. The classifier will be trainable just on positive data, i.e. only non-anomalous traffic and trivially labelled.

However an issue arises: SVMs work on single vectors during the classification process, this means that the packet flow must be compressed and abstracted into one single vector, containing good features to discriminate the normal flow pattern from the anomalies. In other words, we need to implement a mapping like:

$$w = [p_1, p_2, p_3, \cdots, p_i]$$
$$f(w) \to v$$

where $p_i$ is a packet, $w$ a window of packets and $v$ is the resulting flow vector.

We decided to keep a window of the last-minute packets: when a new packet arrives, all the ones older than 1 minute are filtered out. In this stage each packet is abstracted as a vector with the following features:

- source IP address
- destination IP address
- source port
- destination port
- source MAC address
- destination MAC address
- IP protocol
- total packet length
- SYN flag
- ACK flag
- arrival timestamp in nanoseconds

Once the window is updated, the flow vector is computed from it and finally is served to the classifier, which will return a raw estimation that will be classified accordingly to a threshold. The features of the network flow vector are the following ones:

- number of packets
- number of connections
- mean connections per source address
- number of establishing connection
- mean packets length
- mean number of packets per connection

In order to make the whole system testable and usable during development and demo time, we trained the classifier on a small set of dummy data, thus the classifier will detect a ping flooding as an attack.

If the classifier detects an anomaly, i.e. an intrusion, the event is logged to the system journal. A future development of the project could involve desktop notifications thanks to the `notify-send` command.

Since the classification process can be slower than the traffic flow, especially under an attack, we placed it in a separate thread. The main thread of the daemon runs the socket listener and keeps track of the child thread in a variable. The child thread is the one dedicated to window processing, classification and eventual logging. When a new packet arrives, the parent checks the state of the child thread, and if it has completed its run, it

is reinitialized with the new window as input. If the classification thread is busy instead the windows is just updated.

This policy ensures that the system will not saturate its resources running multiple concurrent instances of classifier which will very probably return the same output.

# 5 Firewall

The second key feature is filtering the incoming traffic according to custom rules. Those rules have to be stored in a `rules.json` file and in order to be applied the user space daemon must be restarted. We believe that having the rules in a file makes it easier to version them, also if they have to be applied across system replicas.

Once the rule file is configured, the userspace daemon has to be restarted in order to re-parse the file and dispatch the rules to the kernel module, which will enforce them on the incoming traffic.

## 5.1 Rule Format

The rule specification file is composed of three properties:

**Lock** If set to true all the incoming traffic will be dropped, bringing the system into a lock-down mode. This option is meant to quickly stop network attack and intrusions and also to slow down attackers.

**Rules** An array of rule objects. Each rule has four fields:

- source: the packet source IP, "*" can be used as wildcard.
- destination: the packet destination IP, "*" can be used as wildcard.
- port: the packet destination port, 0 can be used as wildcard.
- protocol: the packet protocol (like 6 for TCP), 250 can be used as wildcard since it doesn't refer to any protocol.
- action: the action to apply if the rule matches. 0 is used to accept the packet, while 1 is used to drop it.

**Default** the default policy to apply if the packet matches no rule.

### 5.1.1 Rule file example

Assuming the existence of an SSH server running on port 22, the following example configuration file denies SSH connections and allows everything else:

```
{
  "lock": "false",
  "rules": [
    {
      "source": "*",
      "destination": "*",
```

```
      "port": 22,
      "protocol": 6,
      "action": 1
    }
  ],
  "default": "ACCEPT"
}
```

## 5.2 Rule Enforcing

Inside the kernel module, the lock flag and the default policy are stored inside global variables, while the list of rules is stored inside a linked list of `struct rule`.

```
typedef struct rule_struct {
  char source[16];
  char destination[16];
  unsigned short port;
  unsigned char protocol;
  char action;
} *rule;

typedef struct rule_list_struct {
  rule data;
  struct rule_list_struct *next;
} *rule_list;

static int LOCK = TC_ACCEPT;
static int DEFAULT = TC_ACCEPT;
```

Once the userspace daemon connects, the old rules are removed and the new ones get applied.

```
static void set_netlink_pid(struct sk_buff *skb){
    // ...
    nlh = (struct nlmsghdr *)skb->data;
    payload = (char *)nlmsg_data(nlh);
    msg_rule = payload + 5;

    if (strncmp(payload, "RULE", 4) == 0) {
        if (msg_rule[0] == '0') {
            msg_rule++;
            if (strncmp(msg_rule, "LOCK", 4) == 0) {
                LOCK = TC_DROP;
                clean_rule_list();
            } else if (strncmp(msg_rule, "UNLOCK", 6) == 0) {
                LOCK = TC_ACCEPT;
                clean_rule_list();
            } else if (strncmp(msg_rule, "DEFAULT", 7) == 0) {
                DEFAULT = strncmp(msg_rule + 8, "DROP", 4) == 0 ?
   TC_DROP : TC_ACCEPT;
            }
        } else {
            r = (rule) (msg_rule + 1);
            create_rule(r);
            print_list();
        }
```

```
    }
}
```

Once the kernel receives a packet, the lock is immediately checked: if it is activated, the packet is dropped and not even dispatched to the user space. The logic behind this behavior is that we are reacting to an emergency situation with a drastic countermeasure and thus we already know we are under attack, a classification will just steal system resources.

```
    if (LOCK) {
        return NF_DROP;
    }
```

If the incoming traffic is instead unlocked, the rules linked list is read linearly. Like in `iptables`, the rule order is essential, both for correctness and efficiency. If the packet matches a rule, the corresponding action is returned and it's forwarded to the user space classifier only if accepted.

```
unsigned int match_rules(struct ethhdr *ether, struct iphdr *iph, struct
    tcphdr *tcph) {
    // ...
    while (curr != NULL) {
        rule rule = curr->data;

        source_match = strncmp(rule->source, "*", 1) == 0
            || strncmp(rule->source, packet_source, 16) == 0;
        dest_match = strncmp(rule->destination, "*", 1) == 0
            || strncmp(rule->destination, packet_dest, 16) == 0;
        port_match = rule->port == 0
            || ntohs(tcph->dest) == rule->port;
        proto_match = rule->protocol == 250
            || (char)iph->protocol == rule->protocol;

        if (source_match && dest_match && port_match && proto_match) {
            printk(KERN_INFO "Packet matched Rule %d\n", index);
            return rule->action;
        }
        curr = curr->next;
        index++;
    }
    return TC_NO_MATCH;
}
```

If no rule matches, the default policy is applied: in this case even dropped packets are dispatched to the user space.

One may be confused by this procedure, but the reason behind it is very simple. When we explicitly drop a packet with a dedicated rule, we are aware of its malicious presence and is already discarded from the flow. When the default drop rule is invoked instead, the packet was not caught by our rules, this can be consequence of a misconfiguration of the rules. Thus we believe that it should be included in the incoming traffic flow analysis.

# 6 How to run the project

The project release comes with a useful installation script, just run it to get the system configured. You will just have to customize the rules.json file, start the service with:

```
$ sudo systemctl start 2columnS.service
```

And connect to the journalctl to observe the logs with the following command:

```
$ sudo journalctl -f -u 2columnS.service
```

The entire service is manageable with systemctl, so can be restarted with:

```
$ sudo systemctl restart 2columnS.service
```

Stopped with:

```
$ sudo systemctl stop 2columnS.service
```

And checked with:

```
$ sudo systemctl status 2columnS.service
```

# 7 Conclusions

This paper has shown the development of a Machine Learning based tool and simple firewall.

We have faced many challenges, first of all kernel programming. Here we had to be very careful about bugs and efficiency, since every minimal error could break the Virtual Machine OS (and did break our VMs multiple times). Moreover, documentation on this topic is very scarce, and we had to rely on the original source code comments[2].

On the Machine Learning side, it was difficult to retrieve the dataset and find good generalized techniques that were not based on a specific network architecture. This leaded us into proposing a solution that had generalization as main purpose and to composing a small dataset from our own networks.

On hindsight, this project has been very helpful on different levels: it allowed us to put in practice many concepts studied in this course and in other security ones, it was an opportunity to play with the Linux kernel and understand how to navigate its components, hopefully creating something new.

[2]Made easier thanks to Bootlin website

# References

[Du] Wenliang Du. *Firewall Exploration Lab*. URL: https://seedsecuritylabs.org/Labs_20.04/Files/Firewall/Firewall.pdf.

# Evaluation Page

| |
|---|
| **Technical Content [0-10]:** |
| **Deepness And Soundness [0-3]:** |
| **Presentation [0-2]:** |