iplite: a lightweight packet filter for NuttX

Eduardo Menezes Moraes[1], **Rodrigo Teixeira de Souza**[1], **Rafael Oliveira da Rocha**[1], **Lourenço Alves Pereira Jr.**[1]

[1]Divisão de Ciência da Computação (IEC)
Instituto Tecnológico de Aeronáutica (ITA)
12228-900 – São José dos Campos – SP – Brazil

{eduardo.moraes,rodrigo.souza}@ga.ita.br, {ljr,rafaelror}@ita.br

Abstract. *The project proposes a lightweight packet filter in a Real-Time Operating System (RTOS), aiming to provide an additional security layer to embedded systems, allowing the users to create their security policies through the filtering process of the ingress network packets. The* `iplite` *firewall was implemented on NuttX OS based on the best practices of the Linux Netfilter firewall and consists basically of two parts: an application on user space, homonymously called* `iplite`*, which serves to provide the user CLI, besides a module on kernel space,* `netfilterlite`*, responsible for providing the APIs. As an open-source project, our solution allows the reproducibility of the experiments and the firewall core adaptation to other operating systems.*

Resumo. *Esse projeto propõe a implementação de um filtro de pacotes leve para um Real-Time Operating System (RTOS), visando fornecer uma camada adicional de segurança aos sistemas embarcados, permitindo que os usuários criem suas próprias políticas de segurança, através do processo de filtragem de pacotes de ingresso na rede. O* `iplite` *foi implementado no NuttX OS, sendo composto por uma aplicação na userspace, homonimamente* `iplite`*, que serve para fornecer a CLI ao usuário, e um módulo no kernel space,* `netfilterlite`*, responsável por prover as APIs. Esse é um projeto de código aberto, permitindo que outros reproduzam experimentos, bem como repliquem conceitos utilizados desse firewall em diferentes sistemas operacionais.*

1. Introduction

Nowadays, there is a tendency for the internet of things to be increasingly more present in people's lives, even intensified by the COVID-19 pandemic — there is no lack of estimates that indicate a relevant increase in its usage in the coming years [Wegner 2022, Idzikowski et al. 2018]. The rise of those systems came together with the digital transformation as it pervades multiple application domains, from smart homes and smart cars, in the medical, to industrial and public institutions. However, cybersecurity is not always able to keep up with this growth. Hence, IoT devices can have serious security issues since they usually have weak update cycles and poor maintenance, as seen in some malware attacks such as Mirai [Chacos 2016] and Mozi [McMillen 2021].

Besides, researchers reported in one study [Cui et al. 2009] that embedded devices were over 15 times more vulnerable to Internet-based threats than enterprise networks,

whereas computational constraints suppress robust solutions [Gayle 2016]. Despite consolidated firewalls in the open-source world, such as Netfilter, lightweight firewall solutions for low-power devices remain an open issue. Thus, given the difficulty of finding an open-source firewall for embedded systems, we propose the `iplite` to fill this gap.

Therefore, we aim to develop a packet filter for NuttX, an embedded RTOS, that filters unauthorized network packets and prevents malicious traffic from targeting vulnerabilities on embedded devices, focusing initially on connection-oriented transmissions (TCP transport layer). However, the problem with creating a firewall for embedded systems is that it's usually not convenient to use other operating systems' common and popular packet filters. For example, the Linux firewall, composed, respectively, of Iptables and Netfilter in its user and kernel space, was aimed to run in more complex and resourceful hardware. Thus, it probably would not perform satisfactorily in tiny chipsets, including those commonly used in IoT (Internet of Things). Consequently, a light packet filter, carrying only the main commands focused on the needs of a specific embedded design, would be truly useful and welcome for many low-power chipsets, which according to [Niedermaier et al. 2019] represents an important part of the industrial scenario. In that regard, `iplite` is a lightweight firewall for NuttX - based on the Linux firewall's best practices, but using its original functions, patterns, and commands, totally redesigned to facilitate developers to understand how to work with it, besides being lighter than the Netfilter project, attending better to tiny chipset restrictions.

To accomplish the aforementioned, the `iplite` was divided into two parts: the userspace app, the `iplite`, and the kernel space module, the `netfilterlite`. The `iplite` is the CLI responsible for receiving the user input, which is a command specifying the packet filtering rule composed of IP addresses, TCP ports, and the rule. Moreover, the `netfilterlite` will provide the APIs that are used by the userspace tool and store the user-specified rules in a chain table in the kernel space.

In the next sections, we will go through the architecture and main functionalities of our packet filter, as well as some fundamentals, and basic concepts that are essential for understanding our project. Furthermore, a guide on how to configure the environment and install the necessary dependencies for setting up the project and reproducing the experiments will be available, as well as a tutorial video with a demonstration of the tool. Then, the next sections will be divided in the following way: project architecture and main functionalities, experiments, and conclusion.

2. Architecture and Main Functionalities

2.1. `iplite`

`iplite` is a utility program that allows users to configure the packet filter rules. The user interacts with it via NuttX shell CLI by adding the rule followed by a tuple composed of the source IP address, source TCP port, destination IP address, and destination TCP port. The input rule is passed to `netfilterlite` module, which creates a new entry on the chain rule. In this initial version, `iplite` applies only to the IPv4 protocol.

2.1.1. Implementation

The `iplite` is a CLI utility that receives five arguments in the following order: the filtering rule, source IP address, destination IP address, source TCP port, and destination TCP port, as shown in Figure 1.

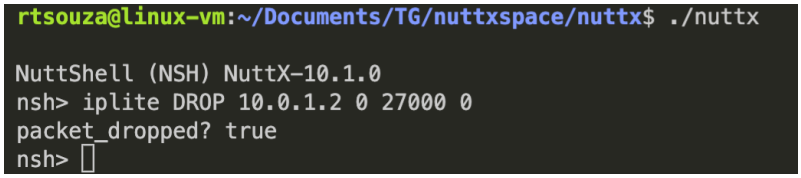

Figure 1. `iplite` call.

Besides receiving user input, `iplite` translates this entry data into a proper structure used in the kernel space, as shown in some code snippets in Figure 2.

The filtering rule input (DROP/ACCEPT) is translated to an equivalent integer used in the `netfilterlite` module by processing the received text and associating an enumerable to it when the input is valid. This enumerable is defined on the `netfilterlite` module as a `rules` type, being 0 associated to the DROP rule and 1 to the ACCEPT rule.

On `netfilterlite` module, the IP addresses are represented as `in_addr_t`, a structure commonly used on C/C++ socket programming for representing IP addresses as a 32-bit unsigned integer. This conversion of IP address format from text to binary is done by a socket API function called `inet_pton`, which receives arguments in the following order, converting the IP address to little-endian: specifier of the family of the address (`AF_INET` or `AF_INET6`), IP address in text format, the variable reference used to save the converted IP address.

Yet on `netfilterlite` module, the transport layer ports are represented as `in_port_t`, another structure used on C/C++ socket programming, commonly used for representing TCP ports as a 16-bit unsigned integer. The socket API function called `htons` implements the endianness conversion and translates an unsigned short integer into the network byte order. This function receives as an argument the unsigned short integer to be put into network byte order and returns the translated short integer. More details about the used network structures and functions mentioned can be found in its library documentation, `arpa/inet.h` [TheOpenGroup 1997].

Obtained the `netfilterlite_addrule` arguments, now we can add the rule to the `netfilterlite` chain rule.

2.2. `netfilterlite`

`netfilterlite` is a framework that allows some networking-related operations and functions for packet filtering and also provides the functionality required for directing packets through a network and prohibiting packets from reaching sensitive locations within a network. This lite adaptation of Netfilter implements DROP and ACCEPT operations, serving as a simple firewall for blocking and allowing specific traffic by IP address and TCP port, focusing on the TCP/IP stack and connection-oriented transmissions (TCP transport layer).

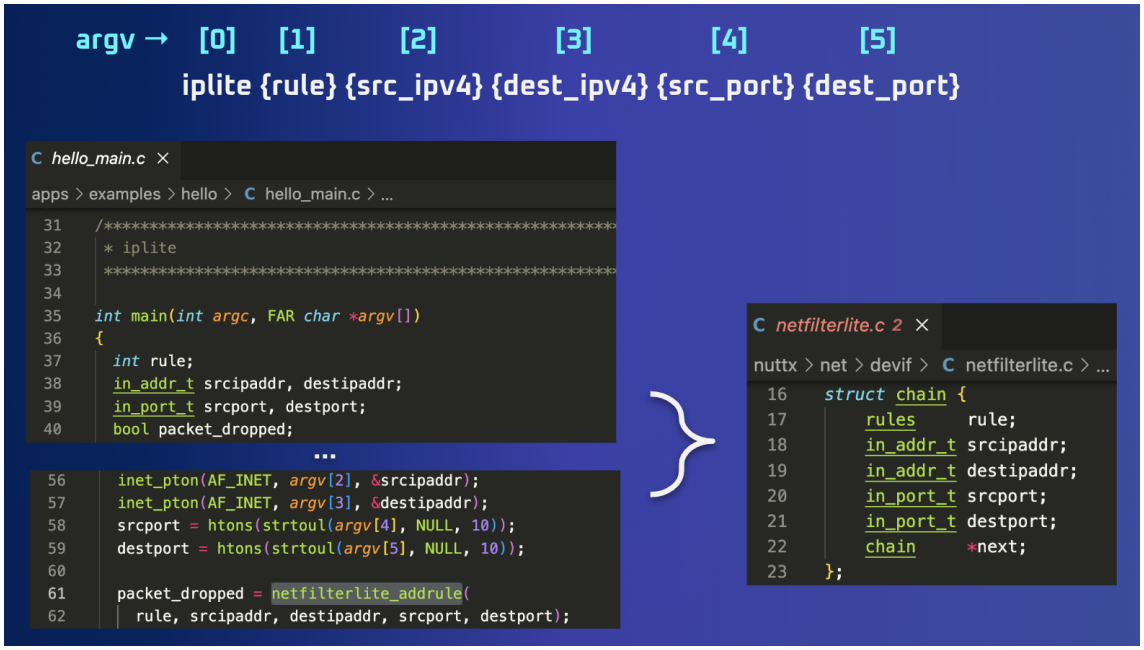

Figure 2. Snippets of the `iplite` implementation.

2.2.1. Main structure

`iplite` rules are structured similarly as in Iptables. First, a table stores all the rules, and each row is a linked list, where a node represents the packet data to be filtered. The action rule (DROP or ACCEPT) and four-tuple (source IP, source TCP port, destination IP, destination TCP port) identify each chain rule. Those chain rules are loaded in the kernel memory space for later use in the filtering process, done by the `netfilterlite` module.

The Netfilter is the packet filter used to base the `netfilterlite` implementation conceptually. It is composed of a chain rule implemented using a linked list, where each node on the chain carries the rule data, and an API set to support Iptables user demands. This way, similarly to the Netfilter chain rule structure, our data structure for storing the packet filter rules is a singly linked list of `chain` nodes. In our implementation, there are two references used in our linked list structure: `chain_head` and `last_rule`. The `chain_head` variable is a reference to the beginning of the linked list, and the `last_rule` variable is a reference to the end of the linked list used to add rules to the chain in constant time.

2.2.2. APIs

The `netfilterlite` has APIs that are used along NuttX kernel and in `iplite`. They are used for initializing the needed data structures, adding a rule to the chain rule, and verifying if a packet is valid or not. Those functions are described below.

`netfilterlite_initialize`

This function is used for initializing the data structures of `netfilterlite`

module. It consists of creating a sentinel node for the linked list head node (named `chain_head`) and initially setting the last node of the chain as the head node. Its implementation is shown in Figure 3.

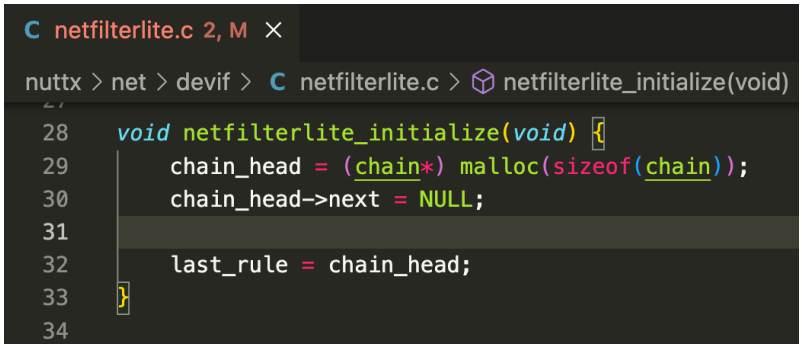

Figure 3. `netfilterlite` initialization function.

`netfilterlite_addrule`

This function is used for adding a rule to the chain rule. It is implemented by adding a new `chain` node to the end of the linked list by receiving the rule parameters as input, creating a new `chain` node with those parameters, and then appending this node to the end of the linked list. In the end, it returns `true` if this process was successful and `false` otherwise. Its implementation is shown in Figure 4.

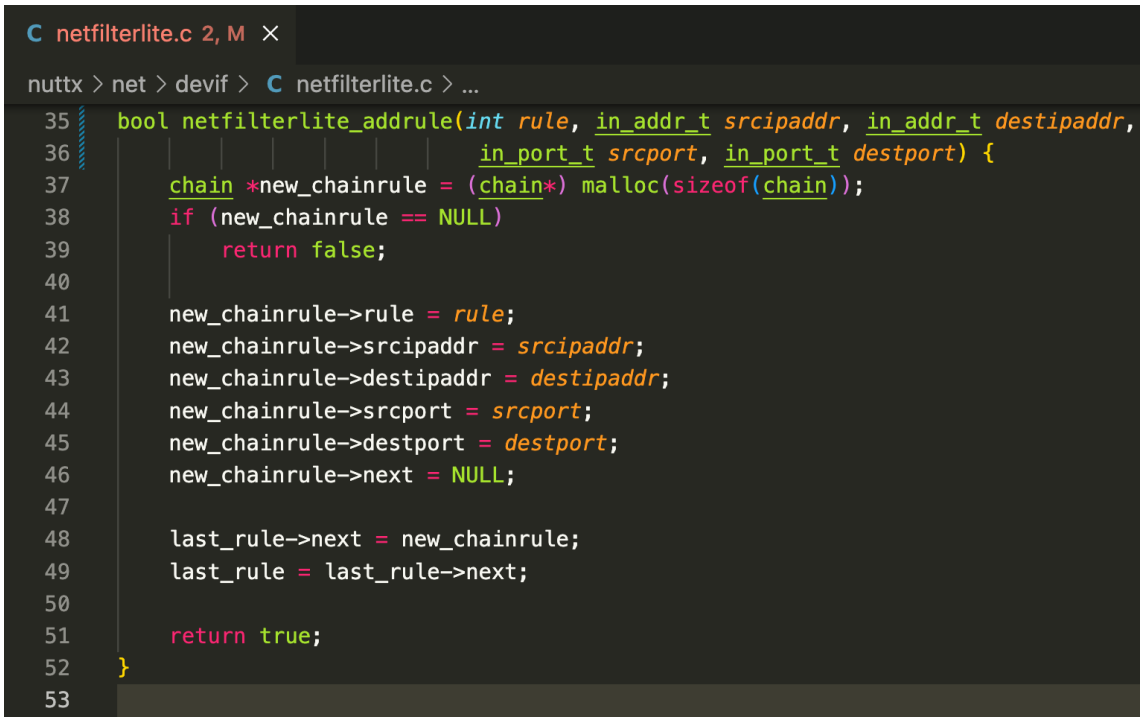

Figure 4. `netfilterlite` function for adding rules.

`netfilterlite_verify_ipv4`

This function verifies an intercepted packet in the ingress traffic by checking its IPv4 header. The device receives the corresponding address reference to that buffer in

memory for each ingress packet and uses it to access the TCP and IPv4 headers. With those headers, we can get information about destination and origin IP addresses and TCP ports for further verification. With that information, we traverse the chain rule linked list, looking for some match between the received info and the existing ones on the chain rule. If a match happens, the packet needs to be dropped since this first implementation only verifies DROP rules, and the function returns `false`; otherwise, it returns `true`. Its implementation is shown in Figure 5.

```
C netfilterlite.c 2 ×
nuttx > net > devif > C netfilterlite.c > ...
77  bool netfilterlite_verify_ipv4(FAR struct net_driver_s *dev) {
                                        ...
91      destipaddr = net_ip4addr_conv32(ipv4->destipaddr);
92      srcipaddr = net_ip4addr_conv32(ipv4->srcipaddr);
93      srcport = tcp->srcport;
94      destport = tcp->destport;
95
96
97      chain *current_rule = chain_head->next;
98      while(current_rule) {
99          /* Verify incoming packet source and destination IP addresses */
100         if (current_rule->destipaddr == destipaddr || current_rule->srcipaddr == srcipaddr)
101             return false;
102         /* Verify incoming packet source and destination ports */
103         if (current_rule->destport == destport || current_rule->srcport == srcport)
104             return false;
105         current_rule = current_rule->next;
106     }
107
108     return true;
109 }
```

Figure 5. `netfilterlite` function for verifying IPv4 header of a packet.

3. Experiments

In order to simplify the `iplite` prototype development, we used the NuttX simulator, a regular program on Linux that simulates an embedded system running NuttX OS, to avoid the necessity of hardware, such as a microcontroller. Thus, it eases the initial implementation and use of the tool. This way, to validate the packet filter's functionality, we have to test if the data structures are initializing, the rules are being created, and taking effect.

3.1. Adding rules

Before we add a rule, let us verify that the NuttX simulator and the Linux machine can communicate. For this, we will do two tests: a ping between one machine and another and opening a TCP/IP connection with an open channel for communication, with the Linux machine being the server.

Initially, before adding the rules, we sent a ping from NuttX (`10.0.1.2`) to Linux (`10.0.2.4`) and then from Linux to NuttX, in order to test the connection on both directions. From the Figure 6 we can see that `ping` works bidirectionally.

Having verified that the communication using `ping` was successful, we will try to block it by adding address filters. For this, we will use `iplite` to add rule shown in

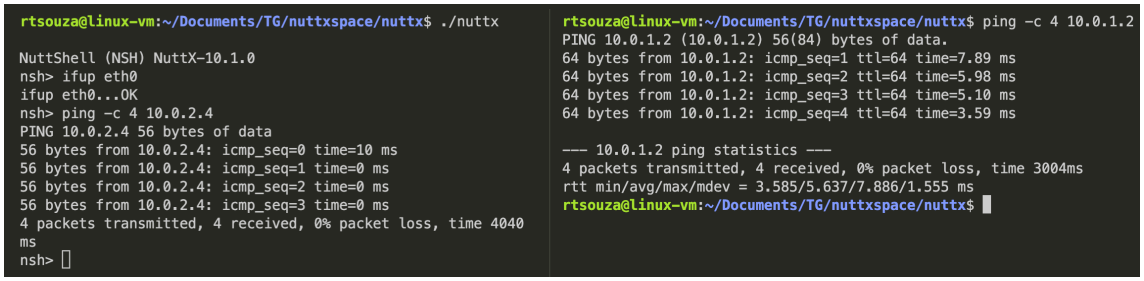

Figure 6. Ping test before address filter.

Figure 7. Notice on the `iplite` command that arguments with value 0 are unspecified, meaning that we are not applying the filter for those arguments.

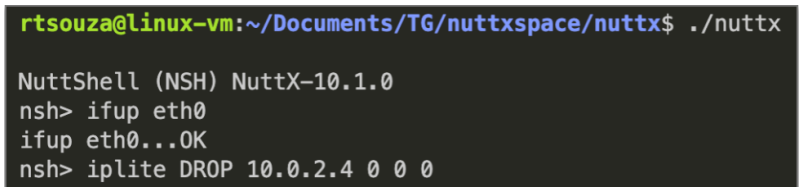

Figure 7. Application of address filter.

3.2. Verifying packet filtering

Added the rule; now we have to check if the packet filtering. To test the packet filter coming from the IP address `10.0.2.4`, we send a ping from Linux to NuttX and vice versa and observe the result. Note from Figure 8 that in both cases, all ICMP echo requests do not have a corresponding ICMP echo reply, causing the ping to relate 100% packet loss.

Figure 8. Pings triggered after IP address filter application.

4. Conclusion

This work presented the `iplite` packet filter and showed its implementation and experiments using it.

After some experiments using a prototype version of `iplite`, it is clear that the project implementation is plausible. However, it can be made in a more sophisticated way, using a complete parser on `iplite` user layer application and allowing the `netfilterlite` to work in other different settings.

For future works, we plan to update `iplite` userspace app to parse more complex commands, including adding and removing rules by the IP address and port,

egress/ingress traffic filtering, and support of UDP transport protocol to allow UDP port filtering. Consequently, the `netfilterlite` will receive the support to handle these other different scenarios, providing more complex APIs to satisfy the userspace needs.

Our project is available in the following GitHub repository: `https://github.com/c2dc/iplite-sbseg2022`

The project manual and instructional videos are available at the following link:

`https://drive.google.com/drive/folders/1e99Oy9Wesl_QHDRpuwlZAygaNjsgaq6_`

References

Chacos, B. (2016). Major ddos attack on dyn dns knocks spotify, twitter, github, paypal, and more offline. `https://www.pcworld.com/article/410774`. Published 21 Oct 2016; accessed 08 Aug 2022.

Cui, A., Song, Y., Prabhu, P. V., and Stolfo, S. J. (2009). Brave new world: Pervasive insecurity of embedded network devices. In Kirda, E., Jha, S., and Balzarotti, D., editors, *Recent Advances in Intrusion Detection*, pages 378–380, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gayle, D. (2016). 'smart' devices 'too dumb' to fend off cyber-attacks, say experts. `https://www.theguardian.com/technology/2016/oct/22/smart-devices-too-dumb-to-fend-off-cyber-attacks-say-experts`. Published 22 Oct 2016; accessed 08 Aug 2022.

Idzikowski, F., Chiaraviglio, L., Liu, W., and van de Beek, J. (2018). Future internet architectures and sustainability: An overview. In *2018 IEEE International Conference on Environmental Engineering (EE)*, pages 1–5.

McMillen, D. (2021). Minternet of threats: Iot botnets drive surge in network attacks. `https://securityintelligence.com/posts/internet-of-threats-iot-botnets-network-attacks`. Published 22 Apr 2021; accessed 08 Aug 2022.

Niedermaier, M., Striegel, M., Sauer, F., Merli, D., and Sigl, G. (2019). Efficient intrusion detection on low-performance industrial iot edge node devices.

TheOpenGroup (1997). arpa/inet.h - definitions for internet operations. `https://pubs.opengroup.org/onlinepubs/7908799/xns/arpainet.h.html`. Accessed 08 Aug 2022.

Wegner, P. (2022). Global iot market size grew 22% in 2021 — these 16 factors affect the growth trajectory to 2027. `https://iot-analytics.com/iot-market-size`. Published 30 Mar 2022; accessed 08 Aug 2022.