

Packet detection using Kernel Module - Linux

Amanda Le, Phil Neff - Team 57

Winter term 2018

1 Introduction

1.1 Context

Most modern networking applications require the use of TCP/IP(Transmission Control Protocol/Internet Protocol) Model or UDP (User Datagram Protocol). The handling of all network packets in Linux is done at the kernel level. The fundamental data structure in Linux networking code is the SKB(Socket Buffer). Every packet that is either sent or received in Linux does so with the use of this data structure.

With the use of these SKB(s), data packets are sent from the User Layer through the Kernel and with the aid of device drivers are then passed through to the Hardware Level. Conversely SKB's also are used to pass data packets back up from the Hardware Layer through the Kernel and up to the User Level. This 2-way communication using SKB's permits the correct passing of packets from Hardware to User

1.2 Problem Statement

Knowing that all network data packets pass through the kernel and will pass through using the SKB data structure can a simple kernel module be used to detect changes to network activity? This could then be possibly further developed for a security application.

The motivation behind this project, is to build upon skills learned during classes and develop a greater understanding of kernel interaction with networking applications.

1.3 Result

Using TCP/IP Headers it was possible to monitor changes in network activities using a basic kernel module. By monitoring baseline activity it was possible to detect network changes by both standard connections (ie connecting to a website on the internet) as well as detect pinging sent from a foreign host using a bridge type connection.

As will be demonstrated, using dmesg with a kernel module is a simple way to track network activity on the host computer.

1.4 Outline

The rest of the report will be as follows

Section2: Presents background and discovery that is relevant to understanding the basics of Linux Kernel Networking

Section3: Will describe the specifics and basic testing and results

Section4: Will show the QC results versus a commercial packet detector

2 Background Information

In order to understand the goal of the project it is necessary to be able to trace how Linux deals with network information.

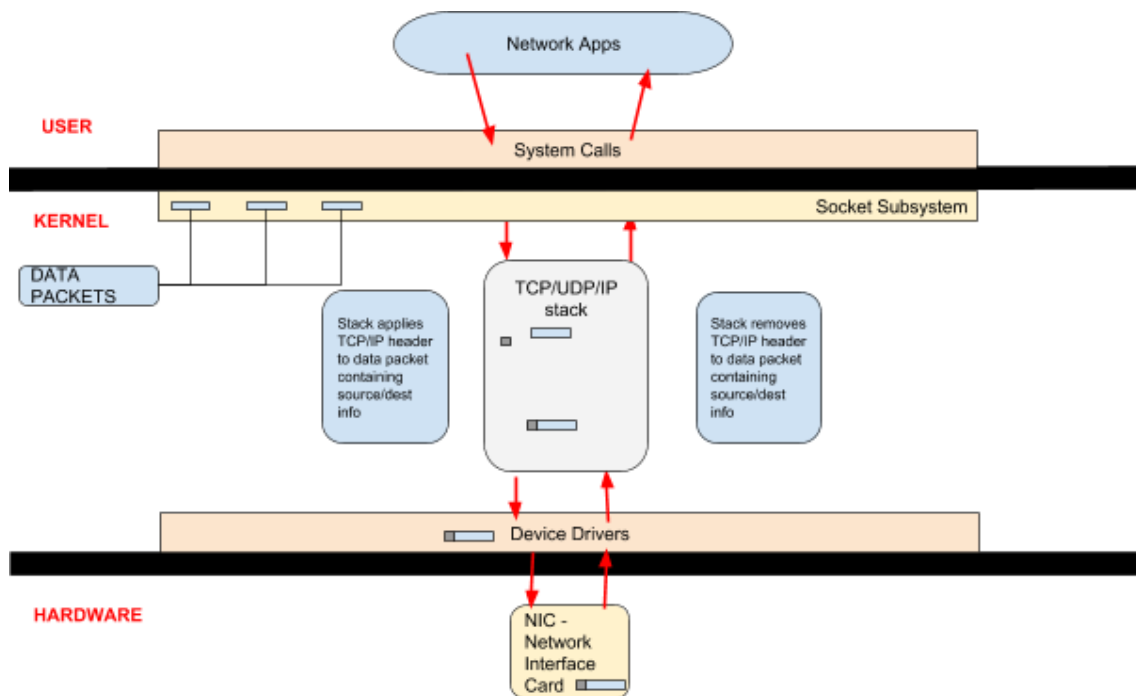


Figure 1: Overview of Network transport through Kernel

Figure 1 - broad overview of how data packets are assigned TCP/IP header and travel through the kernel. The large dark lines represent the various levels within linux. Red arrows indicate information flow with the use of socket buffer data structures. ^[1]

As data is acquired from a network application, system calls are used to direct this data in the Network section of Linux Kernel. Once the security permissions have been checked the

message is then sent to the TCP & UDP stack wherein header and port information for the packets journey are given to data packet.

In order to accomplish this, the SKB structure is used. The SKB is the buffer used in linux for all packets. Pointers to these SKB's are passed in both directions through various functions which are beyond the scope of this project. The structure for the sk_buff is defined within the skbuff.h header file.

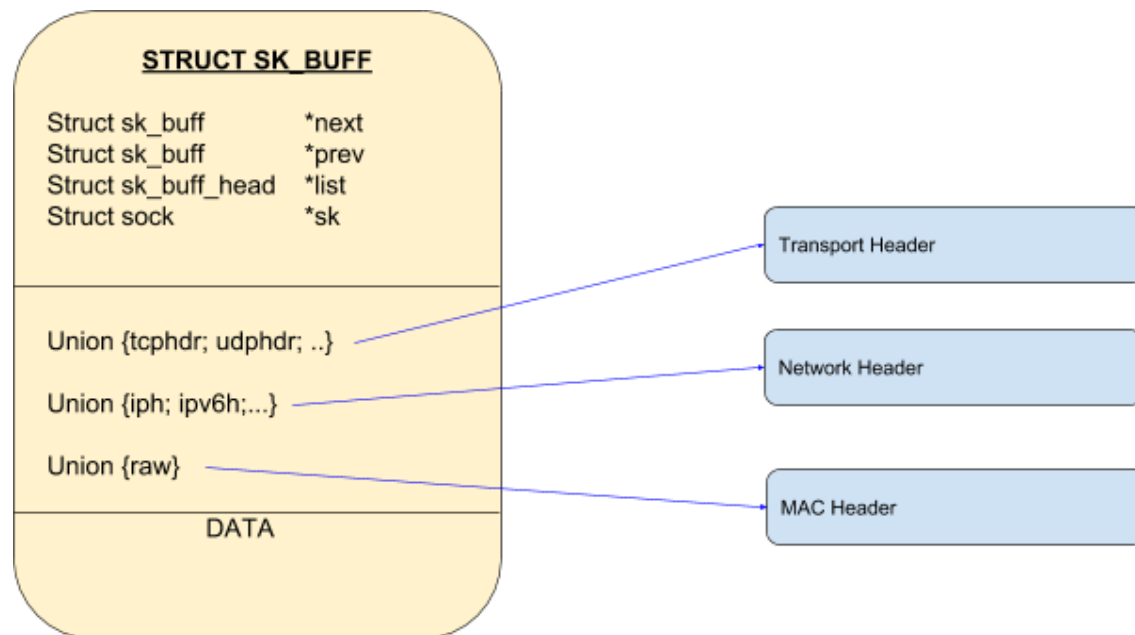


Figure 2: Simplified Struct SK_BUFF

Figure 2 Simplified sk_buff struct. The key points of the structure are easily identifiable. It is possible to join sk-buff in a linked list. There is a pointer to sock struct. As well there all header data is present. Finally the data packet itself is also in the struct. ^[2]

At this point we have a broad overview of how data packets flow through the kernel (figure 1) and how they travel with routing information (figure 2). But, still to discuss is how to monitor once they enter kernel space. To do this, we will use a packet filtering system called Netfilter (netfilter.h & netfilter_ipv4.h).

Netfilter works by having five hook functions which are declared in the ipv4.h. The hooks are designed to be able to analyze packets in 5 pre specified locations on the network stack, thus allowing the user to clearly pick at which point to “hook” a packet. It is important to note that based on return value of the function that is specified by Netfilter. Changing the return value of the function can either accept (allow the packet to continue its trip), drop/steal (packet doesn't continue) or queue. For our project as

we were just monitoring we left this return to NF_ACCEPT, however for fire wall applications the return value could be set to drop or steal to prevent malware if detected.

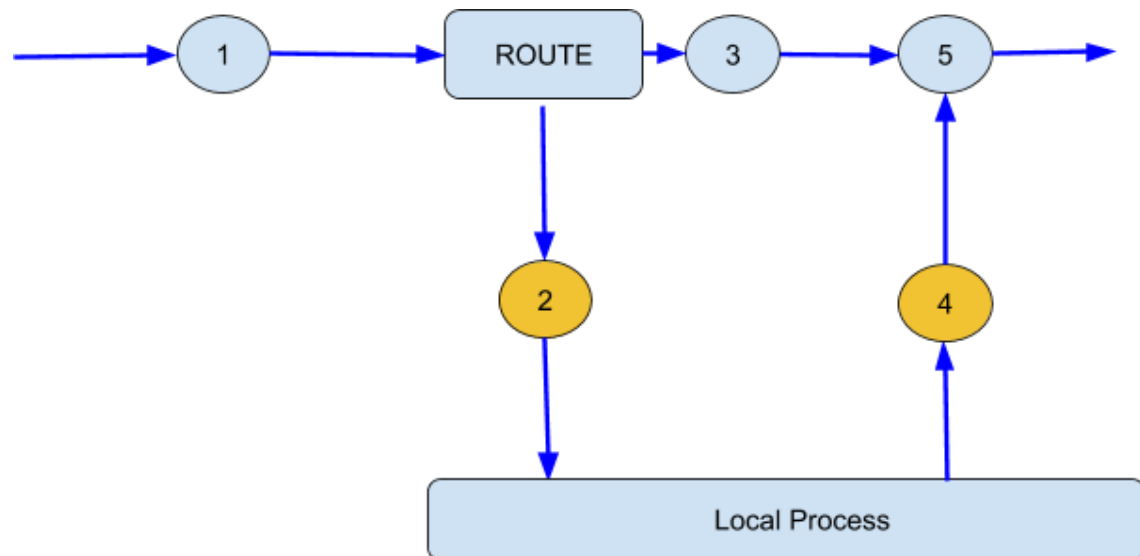


Figure 3: Netfilter hook function positions ^[3]

Netfilter hook function positions^[3]:

1. NF_IP_PRE_ROUTING
2. NF_IP_LOCAL_IN (our hook in)
3. NF_IP_FORWARD
4. NF_IP_LOCAL (our hook out)
5. NF_IP_POST_ROUTING

With these specified positions defined, it is possible to load your predetermined hook functions into the kernel using the insmod command with kernel module. To do this your init_module is configured with a user defined hook function (nfho.hook) that triggers when user specified conditions are met.

The kernel module itself will not be explained in too great a detail as it was covered in class. A simple example was found online and modified to suit our project specifications.^{[4][5]}

3 Result

The kernel module was constructed with 2 hook functions, located at positions 2 and 4 in Figure 3, going into and coming out of the local process. This was deemed sufficient to actually monitor the flow from in/out. Due to the union present in `sk_buff` not all header protocols were monitored, instead it was chosen to only use the predominant types. UDP corresponds to protocol 17 and TCP corresponded to 6.

The module was installed and run under 3 different scenarios to ensure it was working as planned. In order to monitor real time the command: `watch "dmesg | tail -30"` was used for viewing the last 30 lines in the buffer, refreshing every 2 secs. However a full log of all hooks captured is also recorded in the `/var/log` system file. The watch method was used only for demonstration purposes and figures in this document.

Our first use of the module after loading was simply a base case to ensure that being connected to the internet however not using any applications from the User Level we would be able to see only our own IP.

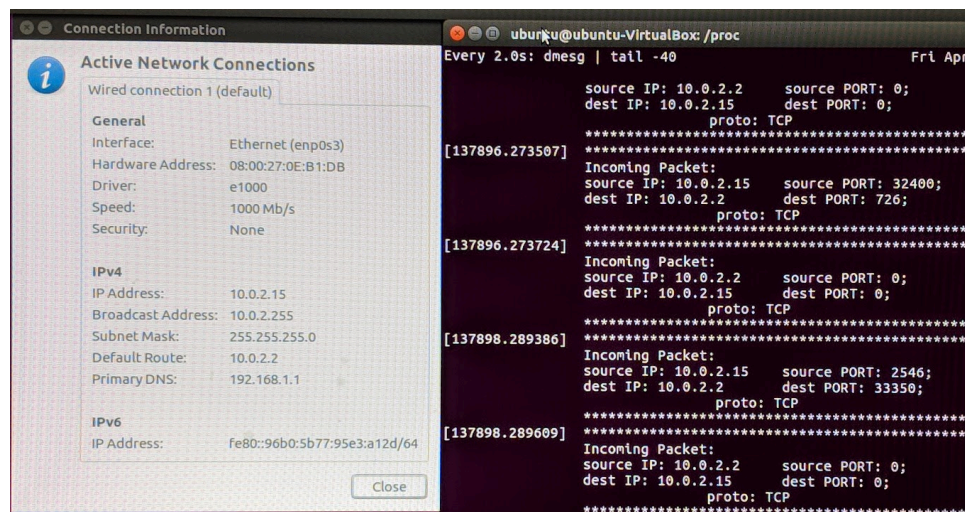


Figure 4: Base case monitoring

Figure 4 showing a side by side with internet information and terminal updating every 2 seconds. Clearly the program shows alternating between IP Address and Default Route as one would expect with no activity from the User Level.

Next was a check to see if our baseline would shift once the User Level initiated an app requiring internet access. Two tests were done on this. The first was done using command line wget command to ensure simplify that only one IP should be entering. The second was loading firefox up to a website. The results are shown below.

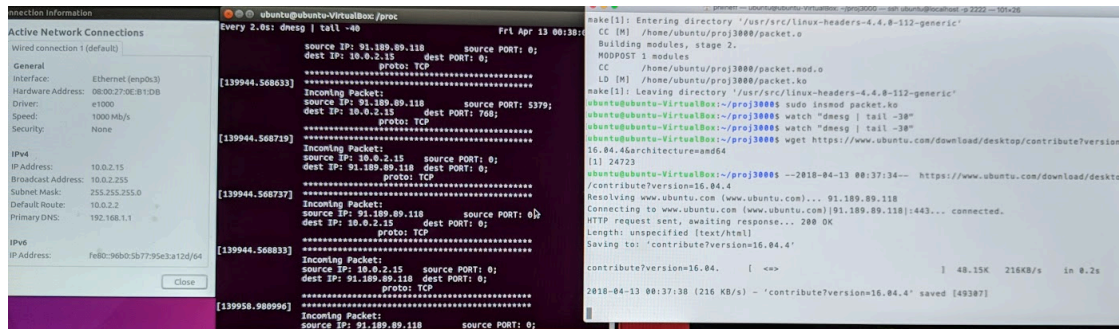


Figure 5: wget command and change in IP addresses being hooked

Figure 5 shows use of the the wget command and the packet transfer that is occurring during a file download. The wget command was used to isolate a single IP address. Opening a web browser also resulted in deflection from our baseline case

Finally a bridging connection was established on the linux machine with the host computer of the virtual box. The host computer then began pinging the virtual machine to see if the IP of the host could be seen.

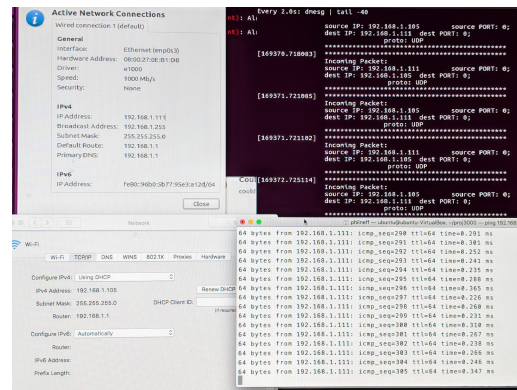


Figure 6: Pinging thru bridge connection

Figure 6 showing host computer with IP of 192.168.1.105 pinging our Linux Machine with address of 192.168.1.111. Also note the UDP protocol is triggered using ping.

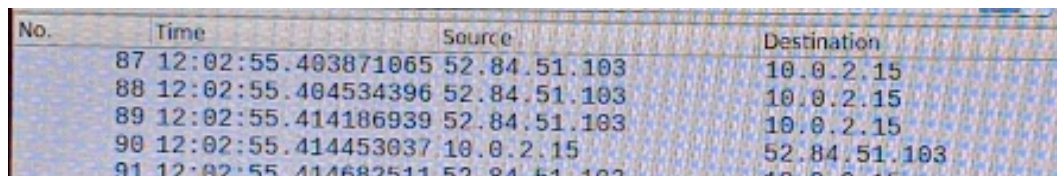
4 Evaluation and Quality Assurance

In order to evaluate our address responses, the program Wireshark was used. Wireshark is a well known cross platform packet detector.

All scenarios were tested using wireshark and our kernel module being run concurrently. In order to get an accurate comparison, it was necessary to examine the system logs located in the folder /var/log/ as seen below.

It is important to specify that the more detailed manner with which to examine packet traffic is the syslog file as it holds the entire record as opposed to the real time monitoring which was for demonstration purposes. An excerpt from the wget download is shown below for an indication of the greater detail that is available using this method.

```
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.083977] *****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056] *****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056] Incoming Packet:
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056] source IP: 52.84.51.103    source PORT: 0;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056] dest IP: 10.0.2.15    dest PORT: 0;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056]                proto: TCP
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084056] *****
*****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084702] Incoming Packet:
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084702] source IP: 52.84.51.103    source PORT: 0;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084702] dest IP: 10.0.2.15    dest PORT: 0;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084702]                proto: TCP
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.084702] *****
*****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094360] Incoming Packet:
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094360] source IP: 52.84.51.103    source PORT: 5635;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094360] dest IP: 10.0.2.15    dest PORT: 768;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094360]                proto: TCP
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094360] *****
*****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094611] Incoming Packet:
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094611] source IP: 10.0.2.15    source PORT: 5891;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094611] dest IP: 52.84.51.103    dest PORT: 768;
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094611]                proto: TCP
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094611] *****
Apr 13 12:02:55 ubuntu-VirtualBox kernel: [ 113.094849] *****
```



No.	Time	Source	Destination
87	12:02:55.403871065	52.84.51.103	10.0.2.15
88	12:02:55.404534396	52.84.51.103	10.0.2.15
89	12:02:55.414186939	52.84.51.103	10.0.2.15
90	12:02:55.414453037	10.0.2.15	52.84.51.103
91	12:02:55.414682511	52.84.51.103	10.0.2.15

Figure 7: Comparison of kernel module vs Wireshark

Figure 7: For the two protocols(TCP and UDP)that were monitored an excellent correlation was seen in packet detection. Source and destination IP's matched with both methods for all test cases performed.

However it was noted that the “all” method contained within Wireshark resulted in more types of packets being observed. This is not a limitation of the hooking method used but rather it reflects the parameters defined and used in the kernel module.

5 Conclusion

5.1 Summary

The main goal of the project was achieved. Our team successfully completed a kernel module that allowed for real time monitoring of network packet transfer through the kernel. Our results provided similar results to proven market packet detectors in various situations.

A greater understanding of networking packet transfer protocol was gained, and then successfully applied using the Netfilter framework to meet the objectives of the project. The project scope was kept purposely confined in order to selectively hook packets.

Real time monitoring could have been done in a more efficient manner(to be expanded on during 5.3 Future Work), however the `dmesg | tail` was sufficient to ensure that the code was correct.

5.2 Relevance

This project took the subject of kernel modules, that was presented briefly during our course and expanded greatly on it. We were able to develop a simple real world application from a concept that was presented in class by applying what we had learned and incorporating new concepts to this.

This project also went a long way to eliminating the unknowns of kernel coding. We cannot walk away from this project and say that “no kernels were harmed in the making of this module”. A lot of reinstallation was necessary to get this done.

5.3 Future Work

This is most definitely something we will continue to improve on. There is a real demonstrated use for what we have made. The basis of what we have done can be used for firewall purposes as well, where rather than simply accepting packets through we could refuse/drop them from unknown IP's or the like.

Being in the IT security stream, this is something we will definitely be updating our resumes with. Rather than simply using a 3rd party software, we now understand exactly what and where the packets we are analyzing are coming from and also at what points we can stop or allow them to pass. None of our previous classes had required us to delve to this level of detail to complete our work.

We would most definitely like to implement a GUI interface for this, wherein the packets are logged and net traffic could then be graphed and presented. Since packet length can be retrieved in the same manner from the SKB this could be included in the outputs as well.

Contributions of Team Members

As we discussed during the project presentation, our final project changed from what we had first proposed. A great deal of time was spent wherein there is nothing to show for it. We both discussed this openly and have agreed that the time spent on each element of our project was 50/50

References

[1] Vandecappelle, Arnout, "networking:kernel_flow [Linux Foundation Wiki]"
https://wiki.linuxfoundation.org/networking/kernel_flow. Accessed: April 1, 2018.

[2] Bratus, Sergey, "The Journey of a Packet through the Linux Network Stack"
http://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Lab9_modified.pdf
Accessed: April 10, 2018

[3] Russell, Paul, "Linux netfilter Hacking HOWTO: Introduction"
<https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO-3.html>
Accessed: April 6, 2018

[4] Evans, Julia, "Day 6: I wrote a rootkit! - Julia Evans"
<http://jvns.ca/blog/2013/10/08/day-6-i-wrote-a-rootkit/> Accessed: Mar 25, 2018

[5] Fontanini Mattias "Programs-Scripts/rootkit/rootkit.c"
<https://github.com/mfontanini/Programs-Scripts/blob/master/rootkit/rootkit.c>, Accessed:
Mar25, 2018

Additionally although nothing was taken, borrowed or changed we would like to reference the following as this series was instrumental in actually conveying what each parameter, function actually was doing. We watched each video a multitude of times so feel that a tip of our hats must be given.

Kankipati, Kiran, "Videos :: The Linux Channel"
<http://the-linux-channel.the-toffee-project.org/index.php?page=videos> Accessed: Mar 20, 2018