European Journal of Interdisciplinary Research and Development
Volume-05 July-2022
Website: www.ejird.journalspark.org ISSN (E): 2720-5746

NETWORK TRAFFIC ANALYSIS AND IP PACKET PROCESSING MONITORING IN LINUX OS

Tadjiev R.N.,
National University of Uzbekistan named after Mirzo Ulugbek
Republic of Uzbekistan, Tashkent City, University Street, House-4
ruhillo1716@ gmail.com

Esonmurodov S.Q.
National University of Uzbekistan named after Mirzo Ulugbek
Republic of Uzbekistan, Tashkent City, University Street, House-4

Annotation
The article deals with one of the topical problems of information security - the control and filtering of network traffic by retaining network packets. To stop network traffic, the Linux kernel structure is used, which describes the network device, and the structure struct net_device_ops, which lists possible operations on a network device; two functions are also used: ndo_start_xmit to handle outgoing packets and rx_handler to handle incoming packets. Using the structures and functionality of the Linux operating system kernel provides the necessary stability for developing software for analyzing the contents of data transmitted in packets, encrypting and decrypting them, and adapting to user requirements. The proposed method can be used to create a new generation of firewalls that implement deep packet analysis technology, and can also be used as a supplement to existing firewalls.
As soon as the amount of information transmitted through computer networks, the problem of information security in local industries is relevant in local networks remains relevant. One important aspect of information security is to manage and filter network traffic. This task is performed by the firewall, often referred to as Brandmower (BrandMower) or Fearwol (firewall). The firewall must make sure that the firewall keeps network packets through it in order to implement the filter.

Keywords: network traffic filtering, network packet capture, packet sniffing, Netfilter/iptables firewall, network device, socket buffer.

Introduction

When studying the operation of a computer network using the IP protocol and troubleshooting, information about the processing of individual IP packets at a network node is interesting. Tools like Tcpdump allow you to record the passage of a packet through the network interface, but do not provide information about the events that happened to the packet in the OS kernel - fragmentation, firewall processing, and the like. However, there is a need for such information. Today, network traffic analysis is a very broad topic. By "analysis of network traffic" we mean the collective name of technologies and their implementations that allow the accumulation, processing, classification, control and modification of network packets depending on their

content in real time. One of the complicating factors, when considering this issue, is the duality of the development of network traffic analysis tools: on the one hand, this is the development of algorithms and approaches to analysis, on the other hand, the development of software and hardware to effectively solve this problem. In turn, this leads to both confusion in terminology and deliberate manipulation of facts and figures for marketing purposes. In this paper, an attempt is made to reflect both the historical development and the current state of this area from a scientific and applied point of view.

The openness of the Linux OS makes it possible to implement a tool for obtaining information about events occurring in the network subsystem of the OS kernel by modifying the kernel source code. This article describes the creation of such a tool.

1. Network Device

The end point for receiving and transmitting data is a network device. Unlike character and block device modules, the Linux operating system does not create files in the network devices /dev directory. Instead, information about registered network interfaces can be retrieved using the ifconfig command.

Network interfaces <linux /netdevice.H > is characterized by the struct net_device structure defined in the file:

Network Interfaces <linux/netdevice.H> is characterized by the net_devicem structure defined in the file

struct net_device { symbol name[NO NAME]; /* network interface name */

...

int race; /* race number */

...

const struct net_device_ops *netdev_ops;/ * network interface performance table * /

...

rx_handler_func_t __rcu *rx_handler; / * address of the incoming packet handling function * /void __rcu *rx_handler_data; /* pointer to data used by the incoming packet handling function */

...

};

There are many fields in the structure (by structure), but it is enough to use the netdev_ops field to store outgoing packets and use the -rx_handler field to store incoming packets.

The netdev_ops field used to receive outgoing packets is <linux/netdevice.H> and contains the address of the net_device_ops structure defined in the file. The structure consists of the following function pointers that perform the operations defined in the network device:

struct net_device_ops { int (*ndo_init) (struct net_device *dev); /* start device */

void (*ndo_uninit) (struct net_device *dev); /* device startup override */

int (*ndo_open) (struct net_device *dev); /* network interface "refresh" (bring to active state) */

(*ndo_stop) (struct net_device *dev); /* "disable" the network interface (put it inactive) */

netdev_tx_t (*ndo_start_xmit) (sk_buff *skb structure, net_device *dev structure); send package */

...
};

If it is necessary to perform operations to control the respective devices, these functions can be performed by the network device driver developer. The GSS translator is built in such a way that when writing a custom driver, the developer can only perform the necessary operations.

One of the main operations is the transmission of a packet to the network. in the net_device_ops structure, the ndo_start_xmit field contains the address of the function that transfers the packet to the network. As arguments, the function refers to the address of the socket buffer and the structure of the corresponding network device.

The packet buffer operation is a system abstraction and is designed to transfer packets between different levels of system network tasks. The socket buffer is characterized by the structure struct sk_buff , which is found in a file on the system

< linux / sgbuff.h >.

structure sg_buff {

...

struct sock * sk ; /* socket with given socket buffer */

net_device * dev ; /* network device connected to the package */

...

u16 transport_header ; /* level traffic * /

u16 network_header ; / - network jump levels* /

mac_header ; /* channel jump levels*/

..

unsigned character * head , * data ; /* packet block pointer, packet data block pointer* /

...

};

The socket buffer contains the control information of the protocols of different levels of the network model and the indicator of the transmitted data. The data field contains the address of the bytes to be transmitted, and the same analysis of the data allows you to create a next generation network display that implements the technology of in-depth packet analysis.

The Ndo_start_xmit function for sending packets over the network < linux / netdevice . returns a value of type netdev_tx with a different name for the netdev_tx_t list defined in file H>:

enum netdev_tx {

__NETDEV_TX_MIN = INT_MIN,

NETDEV_TX_OK = 0x00,

NETDEV_TX_BUSY = 0x01,

};

This list contains the __NETDEV_TX_MIN value, so the count values are guaranteed to be expressed using the integers specified during compilation. In the event that the function of transmitting packets to the network is normal, NETDEV_TX_OK should return a value. Otherwise, NETDEV_TX_BUSY indicates that the network device was unable to complete the transmission of packets for some reason.

2. Processing IP packets in the network subsystem of Linux OS

Traps must be placed in the network subsystem of the OS kernel to intercept information about IP packet processing events. On fig. Figure 2 shows the interaction diagram of the kernel functions that form the basis for the implementation of the IPv 4 protocol; the implementation of the IPv6 protocol is generally similar [4] . "Functions" prefixed with NF_ **on the diagram** are actually implemented **by the function** nf_hook_slow firewall Netfilter , which is a component of the Linux kernel .

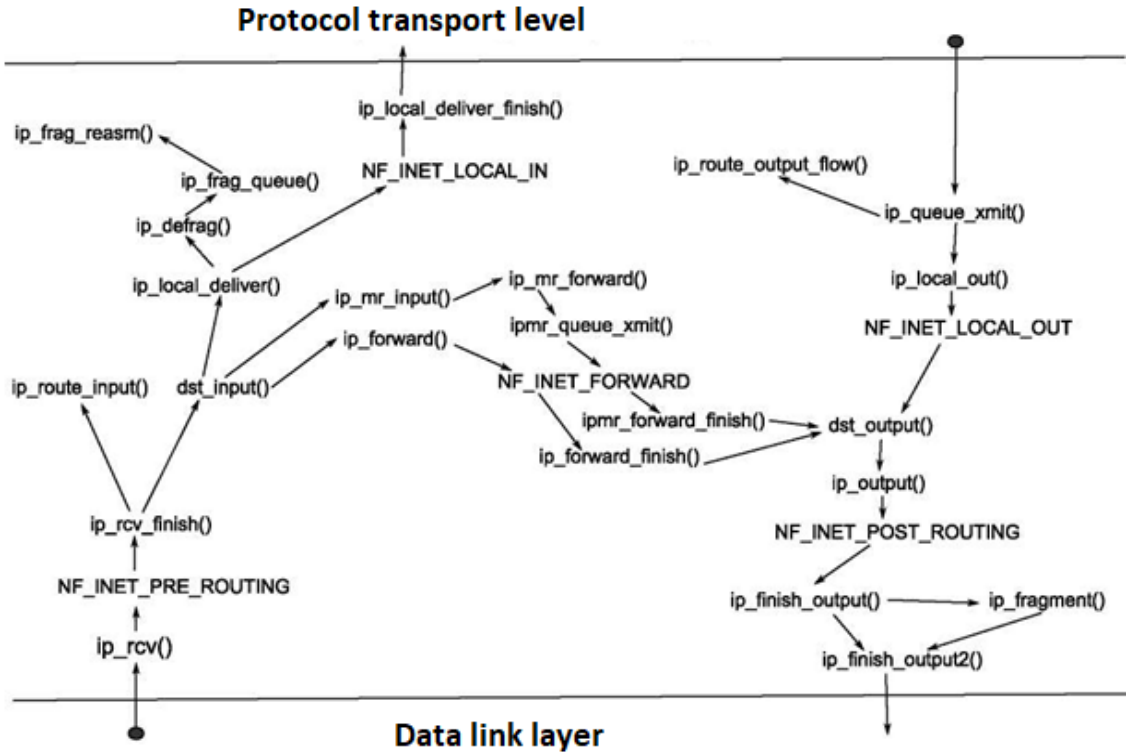

Image. 2**.** Functions of the network subsystem of the Linux kernel

In table. 1 shows the correspondence between IP packet processing events and the functions of the network subsystem of the Linux kernel .

Table 1 Relationship between package processing events and Linux kernel functions

Event	**IPv4 implementation**	**IPv6 implementation**
Formation of a new IP packet by the sending - node	ip_local_out	ip6_local_out
Delivery of the package to destination	ip_local_deliver finish	ip6_input_finish
Sending a packet to a network interface	ip_finish_output2	ip6_output_finish
Packet arrives from network interface	ip_rcv	ipv6_rcv
Packet forwarding (simple and multicast)	ip_forward, ip_mr_forward	ip6_forward, ip6_mr_forward
Packet Drop	any function + kfree_skb	
Packet fragmentation	ipfragment	ip6_fragment
Package defragmentation	ip_defrag, ip_frag_reasm	ip6_frag_rcv, ip6_frag_reasm
Network Address Translation	nf_hook_slow in netfilter	

3. IP Router

This section provides a schematic of a simple router that can forward packets between two network interfaces. It works as follows.

FromDevice Handler with the initialization string *" eth 0"* receives a packet from interface *eth 0* and puts it in the *Classifier handler,* which separates traffic into ARP requests, ARP replies, and IP traffic.

ARP requests will be passed to the *ARPResponder* handler, which will respond to them. The initialization string for this handler will be the following: *ARPResponder (IP _ interface_address_eth 0, MAC _ interface_address_eth 0).*

ARP responses that come to the interface, obviously, were received as a result of some requests. They were created by the *ARPQuerier handler,* and they are sent to him.

IP packets filtered by the *Classifier handler* are sent to the *Paint* handler. *(1),* which "tags" the package with the parameter specified in the initialization line . It can be an integer from 1 to 255. It means the network interface where the packet came from (for the second interface, the *Paint handler will be called (2)).* Now, using the annotation mechanism, any handler will know from which interface the packet came. Next, the packet goes to the *Strip* handler, which "cuts off" 14 bytes from the beginning of the packet (this is indicated in its initialization string "14"), these 14 bytes are unnecessary for IP Ethernet routing header.

Next, the IP packet arrives at the *Check IP Header handler,* which checks the packet for the correct IP address of the recipient, source, calculates the checksum of the packet and compares it with the one in the header. If all checks are successful, the packet is passed to the output interface unchanged, if something is wrong, the packet is discarded.

Next, the packet hits the *GetIPAddress handler,* which copies the destination IP address from the packet's header into its annotation. Redirects the packet unchanged. Then the packet gets into the *LookupIPRoute handler,* which has a routing table as an initialization string. This handler starts branching the tree, sending the packet through a series of other handlers on the first or second interface, or on the Linux TCP/IP stack.

Let's look at one of the branches. After the *LookupIPRoute handler* the packet goes to the *DropBroadcasts handler,* which, based on a flag annotation pointing to broadcast messages for the physical connection from which the packet came, set by the *FromDevice handler,* drops those broadcast packets. Next, the *CheckPaint handler* the annotation is checked, which indicates which interface the packet came from. If it turns out that the packet goes to the same interface from where it came from, then this is not a completely correct situation, ICMP is generated error.

Next, the packet goes to the *IPGWOptions handler* , which processes the Record options. route and timestamp . red black if black record black black route black is enabled, the router "writes" itself into the list of passed routers. Then the black parameter changes Time stamp black , which is needed to measure the time it takes for a packet to travel from one router to another. In case of any errors, ICMP messages are generated. Next, the packet is passed to the *black handler Fix IP src* black with an initialization string that is black IP black - address of the interface through which the packet will be sent .

This handler, provided that the annotation-flag Fix IP Source enabled will fix the source IP address to the IP address of this interface. This flag is set by the *ICMP* handler . *Error* . The

fact is that ICMP errors should come from this router, and the *ICMP handler error* cannot predict through which interface the packet will go to the network, so it simply includes an annotation-flag Fix IP Source , and after the *Lookup handler IP route* will determine the interface through which this package will leave, FixIpSrc will fix the source IP address to the IP address of this interface. Next, the packet is passed to the *Dec* handler *IP TTL* , which subtracts 1 TTL field packet header, and if the received number does not become 0, sends the packet further, and if TTL expired, passes the packet to the *ICMP handler*

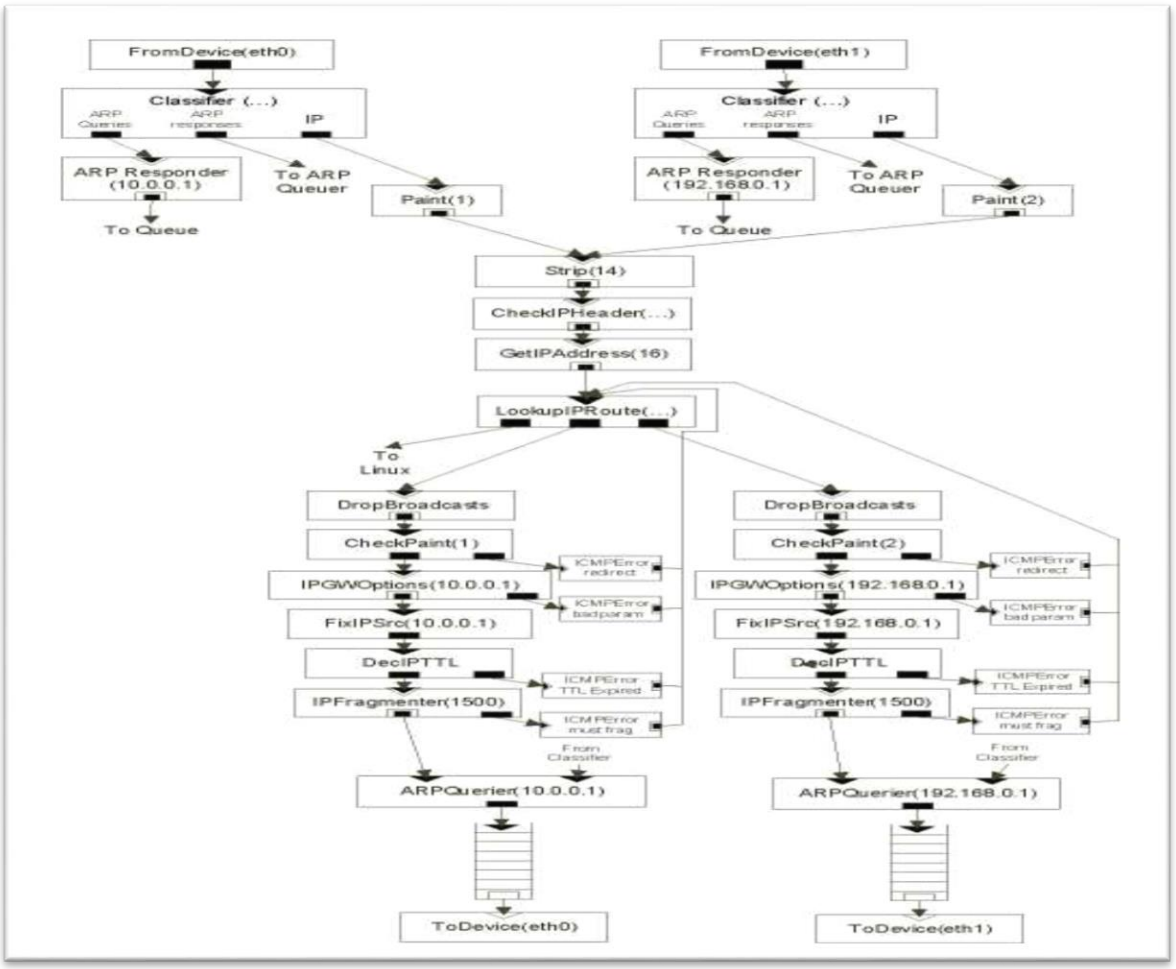

Image. 3. Simple IP router configuration.

error with initialization string *"11"*, which means ICMP error " TTL Expired " which generates a similar error. Packets that successfully pass TTL reduction are sent to the IPFragmenter *handler* with the initialization string *"1500"*. 1500 is the standard MTU for IP networks. Packets smaller than 1500 bytes are skipped by the handler immediately. Those that are larger - it divides into fragments smaller than 1500 bytes, forms new IP packets from these fragments and transfers them to the output interface. However, if the input packet is larger than 1500 bytes and the don't - fragment flag is set in the header, the handler sends the packet to the *ICMP* handler on the second output interface. *error* with an initialization string *of "3,4"*, which means a "fragmentation required" ICMP message, which in turn generates an ICMP packet with a similar error.

After the *IP handler Fragmenter* we receive packets ready to be sent over the link layer. They go to the *ARP handler Querier;* whose initialization string is the IP address of the interface. This handler first receives ARP responses from the *Classifier* handler discussed above. Second, it encapsulates the received IP packets into Ethernet frames and queues them up to be sent to the network interface. This handler also sends ARP requests to the network to recognize the MAC – IP required addresses. As mentioned above, the *ARP handler Querier* using a push connection, "puts" packets into the *Queue with the* initialization string being an integer, which is simply the size of this queue. In this example, the queue is an FCFS structure static size. Frames are "pulled" out of the queue by the to handler using a pull connection. device with an initialization string that is the name of the network interface on the system and sends them to the network as it is freed. This handler works at the MAC / LLC protocol layer. It should be noted that 4 cases were considered above when the packet went to *ICMP error* handlers that, depending on the situation , generated different packets with ICMP errors. All these packets are passed as input to the *Lookup handler IP Route,* and it directs them to the correct interfaces. Below is a figure that visually demonstrates a router, representing it as a connected graph, the nodes of which are the handlers , and the edges are the connections. A schematic representation of this router is shown in image. 3.

Conclusion

The article proposes a universal method for analyzing data of incoming and outgoing network packets using kernel structures.

Modifications of the Linux kernel, allowing you to get complete information about the receipt, processing and forwarding of IP packets. A description of the monitoring tool, based on the described modifications, and an example of its use are given.

The use of kernel functions provides the necessary versatility and the ability to adapt the proposed functions to the needs of the user. The advantage of this method is less dependence on changes in the Linux kernel code. The considered method can be used to create firewalls, including those that implement deep packet analysis technologies.

Literatures

1. Wehrle K., Pahlke F., Ritter H. Linux network architecture. Upper saddle River, NJ, USA: Prentice-Hall, Inc., 2004. 648 p.
2. Olifer V. G., Olifer N. A. Computer networks. Principles, technologies, protocols: textbook. for universities. 4th ed. - St. Petersburg: Peter, 2010. - 113 p.
3. Andrew M White, Srinivas Krislman, Michael Bailey, Fabian Monrose, and Phillip
4. Porras. Clear and Present Data: Opaque Traffic and its Security Implications for the Future. NDSS, 2013.
5. Lavrov A. A., Liss A. R., Yanovsky V. V. Monitoring and administration in corporate computer networks: scientific publication. St. Petersburg: SPbGETU Publishing House "LETI", 2013. 160 p.
6. Chemodurov A.S., Karputina A.Yu. Protection of the Internet gateway and filtering of network traffic of the corporate network // Scientific and methodological electronic journal "Concept". 2015. No. 1. P. 96-100.