INTERCEPCIÓN DE TRÁFICO EN SISTEMAS GNU/LINUX

Jon Ander Ortiz Durántez
jonbaine@gmail.com

20 de Febrero del 2006

La intercepción del tráfico esta considerado por parte de los desarrolladores de IDS´s como uno de los pasos fundamentales para dar solución a los problemas de seguridad actuales, solucionando los problemas que pueden surgir de las respuestas activas (como puede ser el caso de el SnortSam), y dando una respuesta en tiempo real a las necesidades de seguridad reclamadas hoy en día.

ÍNDICE

1. **La importancia de la intercepción de tráfico..1**
2. **Introducción a la gestión de tráfico en sistemas GNU/Linux..................................2**
3. **Primera aproximación a la intercepción de tráfico: A nivel de kernel................. 4**
4. **Modificando iptables desde userland (libiptc)...7**
5. **Comunicación kernel-userland via netlink...8**
6. **Intercepción desde userland: libipq...10**

__________________________________Copyleft_____________________________________

Copyright (c) 2006 Jon Ander Ortiz Durántez
This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.
Copyright (c) 2006 Jon Ander Ortiz Durántez.

Esta obra esta licenciada ba jos los términos de la licencia Atribuciónn-No Comercial Comparte Igual de Creative Commons. Para ver una copia de esta licencia visite http://creativecommons.org/licenses/by-nc-sa/2.0/es/deed.es o escriba una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

1. IMPORTANCIA DE LA INTERCEPCIÓN DE TRÁFICO

La intercepción de tráfico consiste en el análisis del tráfico antes de enrutarlo, es decir, que podemos analizar, manipular, autorizar o denegar el tráfico que pasa, sale o viene a nuestro sistema.

Tradicionalmente se han utilizado otro tipo de librerías para analizar el tráfico de red tal y como llega a nuestro sistema (como por ejemplo Libpcap), pero hay que tener en cuentra que este tráfico se trata de una copia de lo que pasa por la red en ese instante, por lo que no tenemos posibilidad de poder manipular dicho tráfico, sólo podemos analizarlo.

La intercepción de tráfico se utiliza principalmente para sistemas de seguridad y IPS (Intrusion Prevention Systems), la intercepción de echo es lo que marca la diferencia entre los IPS y los IDS, ya que estos últimos sólo pueden detectar los ataque, pero la capacidad de respuesta es bastante reducida, dadas las caracteríasticas de los datos que recibe (Se tratan de copias del trafico de red). Tradicionalmente este tipo de aplicaciones utilizan la respuesta activa (como la solución SnortSam[1] – un output plugin para el Snort),en el mísmo IDS al detectar un ataque, actué sobre el entorno para eliminar esa amenaza (actualizar las reglas de iptables del router para bloquear determinada IP). Este sistema es débil por naturaleza:

- Es débil ante los ataques spoofeados.
- No evita que los paquetes con los que se ha detectado el ataque lleguen a su destino.

Los sistemas que implementan la intercepción de tráfico, solucionan estos problemas:

- Todos los paquetes que son detectados como ataque pueden ser desechados.
- No somos sensibles ante los ataques spoofeados, porque no se modifica el entorno.

La principal contrapartida de la intercepción de tráfico es que se introduce un lag en la comunicación.

La herramienta más conocida que implementa intercepción de tráfico es el IPS Snort-Inline[2], que intercepta tráfico de la librería libipq (en sistemas linux) o de ipfw (en sistemas BSD).

1 http://www.snortsam.net/
2 http://snort-inline.sourceforge.net/

2.INTRODUCCION A LA GESTIÓN DE TRÁFICO EN SISTEMAS GNU/LINUX

Los sistemas GNU/Linux siempre se han caracterizado por ser grandes sistemas orientados (en mayor o menor medida), hacia las redes, esto hace que la gestión adecuada del tráfico de red sea una tarea imprescindible para el kernel.

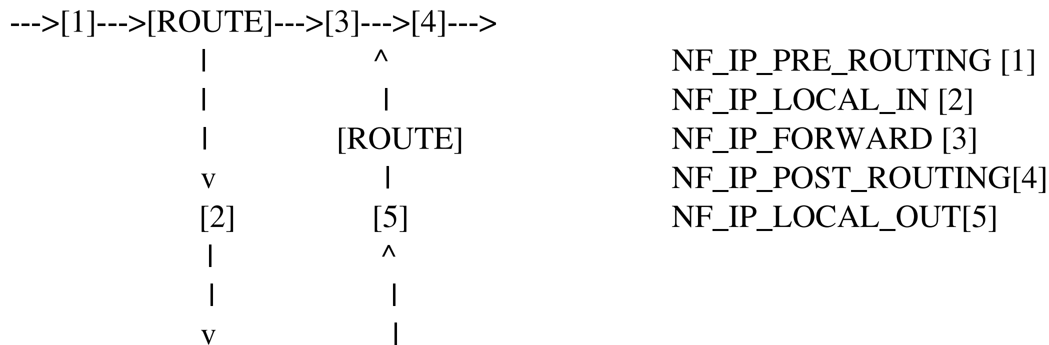
Hoy en día la parte encargada de la seguridad de el tráfico de red que recorre las pilas de los protocolos correspondientes es NETFILTER, existente desde el Kernel 2.4, se trata de una parte del kernel que se encarga de filtrar y dar acceso a los diversos módulos del kernel a las pilas de los protocolos.

Pero antes de NETFILTER, existían otras soluciones, en los kernels 2.2, 2.0 y anteriores, como pueden ser ipchains o ipfwadm (El cual deriva de la solucion ipfw de BSD). El propio Rusty Rusell[3], explica las razones [RR001] de dicho cambio de arquitectura y filosofía de ipchains a Netfilter, entre otras:

- No había una infraestructura para pasar paquetes a espacio de usuario. (Se realizó una solución mediante Netlink, pero era extremadamente lenta).
- No era posible crear reglas de filtrado de paquetes independientes de las direcciones de interfaz. (Había que conocer las direcciones de las interfaces locales para poder distinguir los paquetes).
- La programación del kernel era un cáncer.

¿QUE ES AHORA NETFILTER?

Netfilter es una serie de ganchos (hooks), en varios puntos de la pila de protocolos (No es parte de la pila de protocolos ni nada por el estilo):



Estructura de ganchos de Netfilter dentro de la pila ipV4.

3 Desarrollador de IPCHAINS y NETFILTER (sigue manteniendo actualmente este último en el Kernel 2.6).

El recorrido que siguen los paquetes por la pila del protocolo IPV4 Es el siguiente:

- El paquete entra por la izquierda se realiza sobre el, comprobaciones de seguridad (sanity checks), y entra al primer gancho NF_IP_PRE_ROUTING.
- Pasamos a un proceso de enrutamiento que decide si el paquete viene a un proceso local, o en su lugar es un paquete que hay que enrutar por otra interfaz. En este punto puede haber paquetes que sean desechados porque no pueden ser enrutados.
- Si está dirigido a otra interfaz -> Lo pasa a otro gancho: NF_IP_FORWARD.
- Si es un paquete para un proceso local, pasa a el gancho: NF_IP_LOCAL_IN.
- El tráfico que se genera en los procesos locales pasan por el gancho: NF_IP_LOCAL_OUT, antes de enrutarse bien a un proceso local o a otra interfaz.
- Todo el trafico saliente del sistema, pasa por el gancho NF_IP_POST_ROUTING.

Por lo tanto, Netfilter, proporciona, aparte de otros servicios que luego comentaremos, una serie de ganchos (hooks) a los que diferentes módulos del kernel pueden engancharse (hookearse), para filtrar / manipular ellos mismos el tráfico.

¿Y QUE SON EN REALIDAD LAS IP TABLES?

Las ip tables es un sistema de selección de paquetes sobre el sistema Netfilter. Se trata de un sistema totalmente extensible, cualquier modulo del kernel puede decirle darle a iptables una tabla nueva y decir que los paquetes pasen por la misma.

Internamente está compuesto por tres tablas (filter – filtra los paquetes, mangle – como manipular los paquetes antes de enrutarlos, nat – Para traducciones de direcciones de red).

Los filtros se aplican desde la tabla filter -> solo filtran los paquetes en ningún momento pueden modificar los paquetes. Los filtros se aplican sobre los ganchos NF IP LOCAL IN, NF IP FORWARD y NF IP LOCAL OUT. (Input forward y Output en las construcciones de las reglas de iptables).

Las traducciones o NAT se aplican desde la tabla nat, estas se alimentan de tres ganchos:

- NF IP PRE ROUTING y NF IP POST ROUTING , para alterar el origen y el destino de los paquetes no – locales.
- NF IP LOCAL OUT para alterar el destino de los paquetes locales.

Iptables proporciona un vector en memoria kernel que puede ser leído y modificado desde zona de usuario mediante setsockopt y getsockopt. (para ello probar a hacer un strace en el comando iptables ;)).

3. PRIMERA APROXIMACIÓN A LA INTECEPCIÓN DE TRÁFICO: A NIVEL DE KERNEL

La intercepción de tráfico a nivel de kernel, es adecuada solamente cuando el flujo de paquetes a analizar es muy alto, dado que hoy en día la intercepción de tráfico se puede hacer a nivel de usuario, de una manera mucho más sencilla que programar un LKM.

Para el ejemplo que vamos a realizar vamos a utilizar la interfaz que netfilter provee a nivel de kernel para utilizar los ganchos que tiene sobre la pila de protocolos.

//code

```
#define __KERNEL__
#define MODULE

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>

#include <linux/init.h>
#include <linux/netdevice.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/netfilter.h>
#include <linux/ip.h>
#include <asm/checksum.h> //De aqui utilizamos la ip_fast_csum

static struct list_head lista = {NULL,NULL};
//La lista es solo para funcionamiento interno de la función
static struct nf_hook_ops operations;

static unsigned int operation_hook(unsigned int hook,
                                                  struct sk_buff **skb,
                                        const struct net_device *indev, const
                                        struct net_device outdev, int (*okfn)
                                        (struct sk_buff *))
{

 (*skb)->nfcache=NFC_UNKNOWN;
 printk("Paquete! %d \n",(*skb)->len);
 if ((*skb)->nh.iph->protocol == 0x01){ //Si etamos en el icmph
        (*skb)->nh.iph->protocol = 0x02; //Lo cambiamos =)
        (*skb)->nh.iph->check=0; //Lo ponemos a 0 para calcular ;P
        (*skb)->nh.iph->check =  ip_fast_csum((unsigned char *)(*skb)->nh.iph, (*skb)->nh.iph->ihl);
        (*skb)->nfcache |= NFC_ALTERED; /* Marcamos como modificado ;) */
        return NF_ACCEPT;
}
}
```

```
static int __init init(){
  operations.list = lista;
  operations.hook = (nf_hookfn *)operation_hook;
  operations.pf = PF_INET;
  operations.hooknum = NF_IP_LOCAL_OUT;
  operations.priority = NF_IP_PRI_FILTER-1;
  return nf_register_hook(&operations);
}
static void __exit exit(){
  nf_unregister_hook(&operations);
}

module_init(init);
module_exit(exit);
MODULE_LICENSE("GPL");
```

El ejemplo cambia la cabecera IP de los paquetes icmp que salen de nuestro kernel, a continuación se explican todos los pasos detalladamente.

1) Registrarnos al gancho de netfilter que queramos:

Netfiler exporta la función int nf_register_hook(struct nf_hook_ops *reg) en linux/netfilter.h que permite registrar una función (que será llamada con cada paquete) en un gancho de Netfilter. La estructura que le pasamos tiene los siguientes campos:

```
struct nf_hook_ops
{
        struct list_head list;   //lista par a el funcionamiento interno

        /* User fills in from here down. */
        nf_hookfn *hook;  //Función que queremos que se le llame se trata de un prototipo concreto
        struct module *owner;
        int pf;   //Familia de protocolos
        int hooknum; // que gancho es el que queremos
        /* Hooks are ordered in ascending priority. */
        int priority; //Prioridad para ordenarlos.
};
```

Por lo tanto el código que generamos para registrar nuestra función es el siguiente.

```
operations.list = lista;
  operations.hook = (nf_hookfn *)operation_hook;
  operations.pf = PF_INET;  //Le decimos que familia de protocolos queremos (En este caso familia TCP/IP)
  operations.hooknum = NF_IP_LOCAL_OUT; //Queremos interceptar el tráfico que sale de nuestro sistema.
  operations.priority = NF_IP_PRI_FILTER-1; //Le damos una prioridad
  return nf_register_hook(&operations);
```

2) Escribimos la función para que se la llame desde el gancho:

```
static unsigned int operation_hook(unsigned int hook, //hook de donde viene el paquete
                                   struct sk_buff **skb, //estructura del kernel para manejar los paquetes
                                   const struct net_device *indev, //Por que interfaz ha llegado
                                   const struct net_device outdev, //Por que interfaz sale
                                   int (*okfn)(struct sk_buff *))
```

La estructura que nos interesa se trata una sk_buff (utilizada por el kernel para gestionar los paquetes).

El retorno de la función puede ser varios valores, y le decimos a el kernel que tiene que hacer con el paquete:

NF ACCEPT: continua el recorrido normalmente.
NF DROP: rechaza el paquete; no continúes el recorrido.
NF STOLEN: me hago cargo del paquete; no continúes el recorrido.
NF QUEUE: pon el paquete en una cola (normalmente para tratar con el espacio de usuario).
NF REPEAT: llama de nuevo a este gancho. (Cuidadito con bucles :D)

3) Escribimos la funcionalidad de la aplicación:

```
        if ((*skb)->nh.iph->protocol == 0x01){ //Si etamos en el icmph
                (*skb)->nh.iph->protocol = 0x02; //Lo cambiamos =)
                (*skb)->nh.iph->check=0; //Lo ponemos a 0 para calcular ;P
                (*skb)->nh.iph->check =  ip_fast_csum((unsigned char *)(*skb)->nh.iph, (*skb)->nh.iph->ihl);//Recalculamos el checksum
                (*skb)->nfcache |= NFC_ALTERED; /* Marcamos como modificado ;) */
                return NF_ACCEPT;
```

En este caso lo que hemos realizado es que si encontramos en la cabecera ip en el campo protocolo ICMP (0x01) lo cambiamos por el protocolo 0x02 (IGMP) podéis probar el módulo y capturar el tráfico que generáis con un sniffer a ver que os encontráis =).

El ejemplo que hemos visto es bastante sencillito, pero netfilter está programado de manera que es muy fácil que alguien se construya su propio LKM y aporte al proyecto.

Se pueden también crear hooks de diferentes tipos, y este ejemplo es el más sencillo posible ;) en la bibliografía hay documentos en los que se explica con mas detalle funciones más avanzadas de Netfilter, como los temas de NAT, QUEUE´s...

4.MODIFICANDO IPTABLES DESDE USERLAND (LIBIPTC)

La notificación desde la zona de usuario a la zona del kernel, siempre ha sido un tema delicado (se realiza mediantes get/setsockopt()). Para ello se ha creado una librería que permite que desde zona de usuario se modifique las iptables del kernel.

Esta librería (libiptc) proporciona una serie métodos para recorrer todos los datos de iptables, así como para poder introducir nuevos datos.

Un ejemplo sencillito sería por ejemplo el siguiente que permite listar las cadenas que tenemos dentro de IPTABLES en el kernel:

//compilacion gcc -o iptables iptables.c -liptc

```
#include <string.h>
#include <dlfcn.h>
#include <time.h>
#include "libiptc/libiptc.h"
#include "iptables.h"
int main(void)
{
  iptc_handle_t h;
  const char *chain = NULL;
  const char *tablename = "filter";   //Pillamos de la tabla filter -> podríamos poner tb NAT...
  h = iptc_init(tablename);    //Pillamos la entrada
  if ( !h )   {
     printf("Error initializing: %s\n", iptc_strerror(errno));
    exit(errno);
  }
  for (chain = iptc_first_chain(&h); chain; chain = iptc_next_chain(&h))  {  //las imprimimos todas sin mais :D
    printf("%s\n", chain);  //Imprimimimos solo el nombre...
  }
  exit(0);
} /* main */
```

Si tenemos alguna regla de iptables para filtrar algo, nos daría una salida como ésta:

```
root@JoNaN:/home/jonan/IPTABLES # ./libiptc
INPUT
FORWARD
OUTPUT
```

que serían las cadenas introducidas en el kernel (los lugares sobre los que podemos introducir reglas de iptables). Se podría llegar a mucha más complejidad, pero en la documentación hay muchos ejemplos sobre impresión de todos los valores de iptables.

Las funciones mas importantes de las que nos provee libiptc están perfectamente documentadas en [RR004].

5. COMUNICACIÓN KERNEL-USERLAND VIA NETLINK

Netlink es un sistema de comunicación kernel – userland basado en sockets, mediante dispositivos virtuales. La comunicación recuerda mucho a los Berkley sockets, incluso se pueden hacer comunicaciones unicast y multicast (Formando grupos a los que nos suscribimos). En los sockets Netlink no existen puertos, solamente dispositivos virtuales a los que nos suscribimos, consiguiendo de esta manera una comunicación asíncrona. En un kernel normal hay ya unos cuantos dispositivos creados de antemano:

```
#define NETLINK_ROUTE          0      /* Routing/device hook                    */
#define NETLINK_SKIP           1      /* Reserved for ENskip                    */
#define NETLINK_USERSOCK       2      /* Reserved for user mode socket protocols     */
#define NETLINK_FIREWALL       3      /* Firewalling hook                       */
#define NETLINK_TCPDIAG        4      /* TCP socket monitoring                  */
#define NETLINK_NFLOG          5      /* netfilter/iptables ULOG */
#define NETLINK_XFRM           6      /* ipsec */
#define NETLINK_SELINUX        7      /* SELinux event notifications */
#define NETLINK_ARPD           8
#define NETLINK_AUDIT          9      /* auditing */
#define NETLINK_ROUTE6         11     /* af_inet6 route comm channel */
#define NETLINK_IP6_FW         13
#define NETLINK_DNRTMSG        14     /* DECnet routing messages */
#define NETLINK_KOBJECT_UEVENT  15    /* Kernel messages to userspace */
#define NETLINK_TAPBASE        16     /* 16 to 31 are ethertap */
#define NETLINK_MENDIZALE      17  /*Este es el dispositivo que hemos creado nosotros*/
#define MAX_LINKS 32
```

Pero todos ellos están reservados para un protocolo de comunicación concreto. Por ejemplo el protocolo NETLINK_FIREWALL es el mecanismo que utilizaba libipq para pasar paquetes a nivel de usuario es el NETLINK_FIREWALL, implementar un programa a nivel de usuario que implemente este protocolo, se trata de una dura tarea, dado que tenemos que conocer muy a fondo todo el protocolo de comunicación los cuales no están demasiado documentados.

Si queremos comunicarnos con el kernel (por ejemplo con un LKM de nuestra fabricación), tendremos que insertar un nuevo #define NETLINK_MIPROTOCOLO en include/netlink.h de las sources del kernel, habilitar el soporte a Netlink en el kernel, y recompilarlo para que de esta manera cree el dispositivo virtual y poder comunicarnos mediante él.

La mejor manera de ver un poco su funcionamiento será ver un ejemplillo:

//codigo del programa a nivel de userland

```
#include <asm/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#define MAX_PAYLOAD 1024
struct sockaddr_nl src_addr, dst_addr;
struct nlmsghdr *nlh = NULL;
struct msghdr msg;
struct iovec iov;
int sock_fd;
char * mensaje = "Hola señor kernel!!!";
```

```
int main()
{
        sock_fd = socket(PF_NETLINK, SOCK_RAW, NETLINK_MENDIZALE); //Creamos el socket netlink
        memset(&src_addr, 0, sizeof(src_addr)); //
        src_addr.nl_family = AF_NETLINK;
        src_addr.nl_pid = getpid();
        src_addr.nl_groups = 0;//Sin multicast
        bind(sock_fd, (struct sockaddr*)&src_addr, sizeof(src_addr));  //hacemos un bind (aqui no hay puertos)
        memset(&dst_addr, 0, sizeof(dst_addr));
        dst_addr.nl_family = AF_NETLINK;
        dst_addr.nl_pid = 0; // se lo mandamos al señor kernel
        dst_addr.nl_groups = 0; //sin multicast
        nlh = (struct nlhmsghdr *) malloc (NLMSG_SPACE(MAX_PAYLOAD)); //pillamos memoria
        /* Fill the netlink message header */
        nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);  //La macro NLMSG_SPACE nos da el tamaño
        nlh->nlmsg_pid = getpid();
        nlh->nlmsg_flags = 0;

        strcpy(NLMSG_DATA(nlh),mensaje); //La macro NLMSG_DATA nos da un puntero a la zona de memoria en la que van los datos
        iov.iov_base = (void *)nlh;
        iov.iov_len = nlh->nlmsg_len;
        msg.msg_name = (void *)&dst_addr;
        msg.msg_namelen = sizeof(dst_addr);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;

        sendmsg(sock_fd, &msg, 0); //Enviamos el Hola señor kernel
        recvmsg(sock_fd, &msg, 0); //Recibimos el hola señor usuario :P
        printf("Received message payload:\n%s", NLMSG_DATA(nlh));
        close(sock_fd);
        return (0);
}
```

Y el código a nivel del Kernel en el LKM:

```
//Include de Netlink
#include <net/sock.h>
#include <linux/netlink.h>

#include <linux/module.h>
#include <linux/kernel.h>

 struct nlmsghdr *nlh = NULL;
 struct sk_buff *skb = NULL;
 static struct sock *nl_sk = NULL; //Estructura necesaria para netlink
 int pid; //pid del proceso que escucha :P

static void nl_data_ready (struct sock *sk, int len)
{
         wake_up_interruptible(sk->sk_sleep);
}

int __init init (void)
{
        int err;
        nl_sk = netlink_kernel_create(NETLINK_MENDIZALE, nl_data_ready);//Creamos la historia
        skb = skb_recv_datagram(nl_sk, 0, 0, &err); //Esperamos hasta que se conecte el que nos escucha
        nlh = (struct nlmsghdr *)skb->data;
        printk(KERN_INFO "%s: received netlink message payload: %s\n", __FUNCTION__, NLMSG_DATA(nlh));
        pid = nlh->nlmsg_pid;
        NETLINK_CB(skb).groups = 0;
        NETLINK_CB(skb).pid = 0;
        NETLINK_CB(skb).dst_pid = pid;
        NETLINK_CB(skb).dst_groups = 0;
```

```
        strcpy(NLMSG_DATA(nlh),"Hola señor usuario!!");
        netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT); //Le devolvemos el mensaje
        return 0;
}

void __exit exit(void)
{
        sock_release(nl_sk->sk_socket); //Cerramos el txiringito :P
}

module_init(init);
module_exit(exit);
```

Como habréis podido observar el API es bastante diferente en userland y en kernel, pero en principio la comunicación es muy parecida a los sockets Berkley a los que todos estamos acostumbrados.

El ejemplo es muy sencillote y no explota las características más avanzadas de netlink como puede ser el multicast, para ello debemos ponernos "*apuntarnos*" en un grupo concreto[4]:

```
NETLINK_CB(skb).groups = 1; //Nos apuntamos al grupo 1
```

Y decirle el destinatario (a todos los usuarios de ese grupo), el remitente y enviarlo.

```
 NETLINK_CB(skb).pid = 0;  /* pid = 0 se lo enviamos desde el kernel */
 NETLINK_CB(skb).dst_pid = 0;  /* Queremos que sea multicast */
   /* to mcast group 1<<0 */
 NETLINK_CB(skb).dst_groups = 1; //Grupo destino ->1
       /*multicast a los procesos que escuchan al grupo 1 :D*/
netlink_broadcast(nl_sk, skb, 0, 1, MSG_DONTWAIT); //Enviamos
```

Sobre Netlink no hay demasiada documentación en Internet, y es un tema quizá un poco dejado de lado ya que están apareciendo cada vez más wrapers de protocolos Netlink, ya que estos son un tanto complicados de manejar.

Uno de los ejemplos es el último tema a tratar, que nos permite interceptar tráfico a nivel de userland (Se trata de un wrapper de el protocolo NETLINK_FIREWALL) -> la libipq.

4 OJO!!! este ejemplo sería desde kernel, la API para userland sería distinta.

6. INTERCEPCIÓN DESDE USERLAND : LIBIPQ

Probablemente muy poca gente del mundo GNU/Linux ha oído hablar de la librería libipq, puesto que prácticamente lo único que existe de ella es su página del man.

Para disponer de esta librería programada en C, hay que tener instalado el paquete iptables-dev.

Como ya se ha comentado anteriormente se trata de un wrapper a nivel de userland de el protocolo NETLINK_FIREWALL. Que permite gestionar con total independencia de lo que hay por debajo, que en este caso es una comunicación mediante Netlink.

La librería provee de las siguientes funciones:

```
struct ipq_handle *ipq_create_handle(u_int32_t flags, u_int32_t protocol);
Inicializa la librería y nos devuelve un handle, para hacer después el resto de operaciones.

int ipq_destroy_handle(struct ipq_handle *h);
Libera los recursos sin mais.
int ipq_set_mode(const struct ipq_handle *h, u_int8_t mode, size_t range);
Le decimos a la cola de paquetes el tipo de copia que queremos que nos pase a userland:
        IPQ_COPY_META: Copia sólo los metadatos -> No copia el payload.
        IPQ_COPY_PAQUET: Copia todo el paquete, incluído el payload.

ssize_t ipq_read(const struct ipq_handle *h, unsigned char *buf, size_t len, int timeout);
Leemos de la cola del kernel tantos bytes como le indicamos en len, a un buffer (buff), con un time_out para la operación en milisegundos

int ipq_message_type(const unsigned char *buf);
Determina el tipo de mensaje que hay en el buffer:
        NLMSG_ERROR: Se trata de un mensaje de error de Netlink.
        IPQM_PACKET: Una estructura que lleva metadatos sobre el paquete y opcionalmente el payload.

ipq_packet_msg_t *ipq_get_packet(const unsigned char *buf);
Nos devuelve la estructura contenida dentro de el buffer:
        typedef struct ipq_packet_msg {
                unsigned long packet_id;        /* ID of queued packet */
                unsigned long mark;             /* Netfilter mark value */
                long timestamp_sec;             /* Packet arrival time (seconds) */
                long timestamp_usec;            /* Packet arrvial time (+useconds) */
                unsigned int hook;              /* Netfilter hook we rode in on */
                char indev_name[IFNAMSIZ];      /* Name of incoming interface */
                char outdev_name[IFNAMSIZ];     /* Name of outgoing interface */
                unsigned short hw_protocol;     /* Hardware protocol (network order) */
                unsigned short hw_type;         /* Hardware type */
                unsigned char hw_addrlen;       /* Hardware address length */
                unsigned char hw_addr[8];       /* Hardware address */
                size_t data_len;                /* Length of packet data */
                unsigned char payload[0];       /* Optional packet data */
        } ipq_packet_msg_t;

int ipq_get_msgerr(const unsigned char *buf);
Le preguntamos a instancias superiores que error ha ocurrido -> O_o y nos devuelve una variable de errno

int  ipq_set_verdict(const struct ipq_handle *h, ipq_id_t id, unsigned int verdict, size_t data_len, unsigned char *buf);
Damos un veredicoto sobre el paquete que hemos leido con ipq_read. Opcionalmente podemos también cambiar su payload :P Pero eso sí,
 tenemos que ocuparnos de cambiar sus checksums.
Podemos dar los siguientes veredictos:  NF_DROP ó NF_ACCEPT

char *ipq_errstr(void);
Damos un error  descriptivo de lo que hay en la variable interna ipq_errno
void ipq_perror(const char *s);
Lo mismos pero sin hacer un printf
```

Bueno pues vistas un poco las funciones, necesitamos que alguien en Kernel, apile las llamadas para que lleguen a las colas, y quien mejor para ello que las reglas de iptables (un modulo del kernel también valdría).

Ejemplo:

```
# modprobe iptable_filter    //Cargamos los módulos
# modprobe ip_queue          //Cargamos módulos necesarios
# iptables -A OUTPUT -p tcp -j QUEUE   //Enconlamos todos los paquetes TCP que salgan
```

Ahora tenemos un flujo de paquetes de libipq hacia la librería, pero ¿Que ocurre si encolamos paquetes y no hay nadie escuchando?¿Y si llegan a mayor velocidad de la que la leemos?

Hay una variable interna, que almacena el número máximo de paquetes si sobrepasamos su valor, se comienza a hacer drops, esta variable no decrece hasta que hacemos un set_verdict sobre alguún paquete. (Esta es la mayor pega que se le puede encontrar al asunto).

Por lo demás se puede hacer modulos que intercepten el tráfico eficientemente en espacio de usuario.

//Code -> gcc -o prueba prueba.c -lipq

```
#include <linux/netfilter.h>
#include <libipq.h>
#include <stdio.h>

#define BUFSIZE 2048

static void die(struct ipq_handle *h)
{
	printf("Pos  va y se muere!!");
	ipq_perror("passer");
	ipq_destroy_handle(h);
}

int main(int argc, char **argv)
{
	int status;
	unsigned char buf[BUFSIZE];
	struct ipq_handle *h;

	h = ipq_create_handle(0, PF_INET);	//Pillamos un handle para el libipq
	if (!h)				//sino nos da uno pos es un marron
		die(h);

	status = ipq_set_mode(h, IPQ_COPY_PACKET, BUFSIZE);	//Copiamos el paquete -> Nos da el modo de funcionamiento
	if (status < 0)				//Si no nos deja pos lo petamos
		die(h);

	do{
		status = ipq_read(h, buf, BUFSIZE, 0);	//leemos un buffer
		if (status < 0)
		{
				do{
		status = ipq_read(h, buf, BUFSIZE, 0);	//leemos un buffer
		if (status < 0)
		{
			die(h);
		  break;
		  }
```

```
            switch (ipq_message_type(buf)) {          //Vemos a ver de que tipo es
                case NLMSG_ERROR:                  //Si es un error pos sin mais
                    fprintf(stderr, "Received error message %d\n",
                        ipq_get_msgerr(buf));
                    break;

                case IPQM_PACKET: {     //si es un paquete de ipq
                    ipq_packet_msg_t *m = ipq_get_packet(buf);   //Lo hacemos un ipq_packet_msg_t *m

                    status = ipq_set_verdict(h, m->packet_id,    //decimos que lo aceptamos
                                NF_ACCEPT, 0, NULL);
                    printf("Vamos a ver la longitud  del payload del paquete...\n");
                    printf("%d\n", m->data_len);
                    if (status < 0)
                        die(h);  //esto es si nos da un error al hacer un set_verdict
                    break;
                }

                default:
                    fprintf(stderr, "Unknown message type!\n");   // no conocemos el tipo de mensaje
                    break;
            }
        } while (1);

        ipq_destroy_handle(h);   //Cerramos el handle y a la mierda
        return 0;
}
```

Con éste último método hemos podido ver qué importancia tiene la intercepción en sistemas gnu/linux, y en cualquier sistema en general, viendo además varios métodos para poder implementar un sistema de intercepción tanto en userland como en kernel, así como librerías de apoyo como Netlink y libiptc.

BIBLIOGRAFÍA:

[RR001] ¿Qué hay de malo en lo que teníamos con el 2.0 y el 2.2? Netfilter hacking COMO, Rusty Russell, [01/06/2000]
[RR002]http://www.honeynet.org/reverse/results/sol/sol-21/files/control.c
[RR003]http://lkml.org/lkml/2005/1/1/68
[RR004]http://ldp.rtin.bz/HOWTO/Querying-libiptc-HOWTO/qfunction.html
[RR005]man libipq
[RR006] x-eZine #0 / Art. 008, Raziel, Netfilter.