Sheet 4

sk_buff structure
struct sk_buff

/*
 * Definitions for the 'struct sk_buff' memory handlers.
 * 
 * Authors:
 *  Alan Cox, <gw4pts@gw4pts.ampr.org>
 *  Florian La Roche, <rzsfl@rz.uni-sb.de>
 * 
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

#define SKB_DATA_ALIGN(X) (((X) + (SMP_CACHE_BYTES-1)) & ~(SMP_CACHE_BYTES-1))
#define SKB_MAX_ORDER(X,ORDER)(((PAGE_SIZE<<(ORDER)) - (X) - sizeof(struct skb_shared_info))&~(SMP_CACHE_BYTES-1))
#define SKB_MAX_HEAD(X)  (SKB_MAX_ORDER((X),0))
#define SKB_MAX_ALLOC  (SKB_MAX_ORDER(0,2))

/* A. Checksumming of received packets by device. */

/* NONE: device failed to checksum this packet. */
  skb->csum is undefined. */
UNNECESSARY: device parsed packet and would be verified checksum.
   skb->csum is undefined.
   It is a bad option, but, unfortunately, many of vendors do this.
   Apparently with secret goal to sell you new device, when you
   will add new protocol to your host. F.e. IPv6. 8)

HW: the most generic way. Device supplied checksum of _all_
   the packet as seen by netif_rx in skb->csum.
NOTE: Even if device supports only some protocols, but
   is able to produce some skb->csum, it MUST use HW,
   not UNNECESSARY.

B. Checksumming on output.

NONE: skb is checksummed by protocol or csum is not required.

HW: device is required to csum packet as seen by hard_start_xmit
   from skb->h.raw to the end and to record the checksum
   at skb->h.raw+skb->csum.

Device must show its capabilities in dev->features, set
   NETIF_F_HW_CSUM - it is clever device, it is able to checksum
everything.
   NETIF_F_NO_CSUM - loopback or reliable single hop media.
* NETIF_F_IP_CSUM - device is dumb. It is able to csum only TCP/UDP over IPv4. Sigh. Vendors like this way by an unknown reason. Though, see comment above about CHECKSUM_UNNECESSARY. 8)

* Any questions? No questions, good. --ANK

/ *

#ifdef __i386__
#define NET_CALLER(arg) (*((void**)&arg)-1)
#else
#define NET_CALLER(arg) __builtin_return_address(0)
#endif

struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff *next;
    struct sk_buff *prev;
    __u32 qlen;
    spinlock_t lock;
};

struct sk_buff;

#define MAX_SKB_FRAGS 6
typedef struct skb_frag_struct skb_frag_t;

struct skb_frag_struct {
    struct page       *page;
    __u16             page_offset;
    __u16             size;
};

/* This data is invariant across clones and lives at 
 * the end of the header data, ie. at skb->end. 
 */

struct skb_shared_info {
    atomic_t         dataref;
    unsigned int     nr_frags;
    struct sk_buff   *frag_list;
    skb_frag_t       frags[MAX_SKB_FRAGS];
};

This is a massively important structure. It is the way of representing packets within the kernel. I have deleted some stuff for the purposes of clarity.

struct sk_buff {
    Linking these buffers together. The reason this must be first is that we can cast the packet to sk_buff_head, defined above.
    /* These two members must be first. */
struct sk_buff  *next;  /* Next buffer in list */
struct sk_buff  *prev;  /* Previous buffer in list */
struct sk_buff_head  *list;  /* List we are on */

Back pointer to the sock structure we belong to
struct sock   *sk;   /* Socket we are owned by */

The stamp is the time that the last protocol touched this buffer. Actually, this is a bit more involved than I’m making out – useful for scheduling.
struct timeval  stamp;  /* Time we arrived */

In the administration of network buffers the identity of the device used for sending or receiving the packet must be known.
struct net_device   *dev;   /* Device we arrived on/are leaving by */

Just what you’d expect from a transport layer header, but note the overlay. You'll find the definitions in include/linux/tcp.h::tcphdr, include/linux/udp.h::udphdr, include/linux/icmp.h::icmhdr, etc. So, for example, a udp header is given by:
struct udphdr {
   __u16 source;
   __u16 dest;
   __u16 len;
   __u16 check;
};
/* Transport layer header */

union
{
    struct tcphdr   *th;
    struct udphdr   *uh;
    struct icmphdr  *icmph;
    struct igmphdr  *igmph;
    struct iphdr    *ipiph;
    struct spxhdr   *spxh;
    unsigned char   *raw;
} h;

Again, no surprises here. E.g. from include/linux/ip.h::iphdr we see:

struct iphdr {
    #if defined(__LITTLE_ENDIAN_BITFIELD)
        __u8 ihl:4,
        version:4;
    #elif defined (__BIG_ENDIAN_BITFIELD)
        __u8 version:4,
        ihl:4;
    #else
        #error "Please fix <asm/byteorder.h>"
    #endif
    __u8 tos;
}
__u16 tot_len;
__u16 id;
__u16 frag_off;
__u8 ttl;
__u8 protocol;
__u16 check;
__u32 saddr;
__u32 daddr;
/*The options start here.*/

/* Network layer header */
union
{
  struct iphdr *iph;
  struct ipv6hdr *ipv6h;
  struct arphdr *arph;
  struct ipxhdr *ipxh;
  unsigned char *raw;
} nh;

Still nothing unusual. So e.g. include/linux/if_ether.h::ethhdr

struct ethhdr
{
  unsigned char h_dest[ETH_ALEN];  /* destination eth addr */
This is the control buffer. It is free to use for every layer. Please put your private variables there. If you want to keep them across layers you have to do a skb_clone() first. This is owned by whoever has the skb queued at the moment.

Comment notwithstanding, len holds the length of the packet (including headers), and data_len the length of the data part. csum holds the checksum if it has been calculated. See comment at head of file re checksumming.
Management parameters.

- unsigned int  len;   // Length of actual data
- unsigned int  data_len;
- unsigned int  csum;  // Checksum

This is the length of this buffer, including the length of this struct, used for memory management purposes.

- unsigned int  truesize;  // Buffer size

- unsigned char  cloned,   // head may be cloned (check refcnt to be sure).
- pkt_type,     // Packet class
- __u32          ip_summed; // Driver fed us an IP checksum
- __u32          priority; // Packet queueing priority
- unsigned short protocol; // Packet protocol from driver.
- unsigned short security; // Security level of packet

Actually, see include/linux/skbuff.h::skb_get – this is a reference count to this sk_buff

- atomic_t users;  // User count – see datagram.c,tcp.c

This is a really important bit – it’s where the data resides. The head pointer points to the first part of the buffer (i.e. the bit containing the header), the data pointer points to the part of the buffer containing the data and the tail pointer to whatever follows the data. End, naturally points to the end. There are a lot of helper functions both in this file and in net/core/skbuff.c to allow manipulation of these pointers, the addition of extra space and so forth. See below.

- unsigned char  *head;  /* Head of buffer */
- unsigned char  *data;  /* Data head pointer */
- unsigned char  *tail;  /* Tail pointer */
- unsigned char  *end;  /* End pointer */
void (*destructor)(struct sk_buff *); /* Destruct function */
Sending UDP packets – the code

OK, so let's take a quick look at what happens to the sk_buff when we send a UDP packet (net/ipv4/udp.c). This what gets passed to net/ipv4/udp.c::udp_sendmsg:

```c
int udp_sendmsg(struct sock *sk, struct msghdr *msg, int len)
```

The msghdr here is defined in include/linux/socket.h as:

```c
typedef struct msghdr {
    void      *msg_name;  /* Socket name   */
    int       msg_namelen;  /* Length of name   */
    struct iovec    *msg_iov;  /* Data blocks   */
    __kernel_size_t  msg_iovlen;  /* Number of blocks */
    void      *msg_control; /* Per protocol magic */
    __kernel_size_t msg_controllen; /* Length of cmsg list */
    unsigned   msg_flags;
} msghdr;
```

The data blocks are in an array of iovecs (defined in include/linux/uio.h), each of which is a structure with two fields of interest:

```c
typedef struct iovec {
    void      *iov_base;  /* BSD uses caddr_t, 1003.1g void */
    __kernel_size_t  iov_len; /* Must be size_t (1003.1g) */
} iovec;
```
So, we're being passed an array of pointers to odd bits of data of interest rather than a contiguous area of memory. This is pretty standard within unix. Now, let's go back to the code of `net/ipv4/udp.c::udp_sendmsg`. The next thing of interest is the declaration:

```c
struct udpfakehdr ufh;
```

For this, we need to look [earlier in the file](#) – we see that the first part of this is reserved for a real udp header followed by some other info.

```c
struct udpfakehdr {
    struct udphdr uh;
    u32     saddr;
    u32     daddr;
    struct iovec *iov;
    u32     wcheck;
};
```

The fields of this header are filled in (with the exception of the checksum, which is set to zero) and iov is made to point to the iov we were passed. We then call `net/ipv4/ip_output.c::ip_build_xmit` thus:

```c
err = ip_build_xmit( sk,
                      (sk->no_check == UDP_CSUM_NOXMIT ? udp_getfrag_nosum : udp_getfrag),
                      &ufh, ulen, &ipc, rt, msg->msg_flags);
```

The definition of this routine is below and we care about the first four fields in this context.

```c
int ip_build_xmit( struct sock     *sk,
```
Within this, we have the definition:

```c
struct sk_buff *skb;
```

This next call itself calls `net/core/skbuff.c::alloc_skb`. This allocates a sk_buff from a central store, and initialises it with head=tail=data all pointing to the same allocated block of memory. hh_len essentially represents the MAC header length, rounded up to the next multiple of 16 bytes.

```c
int hh_len = (rt->u.dst.dev->hard_header_len + 15)&~15;
skb = sock_alloc_send_skb(sk, length+hh_len+15, flags&MSG_DONTWAIT, &err);
```

The first call moves the data and tail pointers forward by hh_len, to give us some header room and the second moves the tail pointer forward to give us more data room. When we’re done, iph points to the data part of the structure.

```c
skb_reserve(skb, hh_len);
iph = (struct iphdr *)skb_put(skb, length);
```

We use our callback to get the data out of the fake header we were passed and to stick it in the data part, possibly after an ip header. This is either `net/ipv4/udp.c::udp_getfrag` or `net/ipv4/udp.c::udp_getfrag_nosum`, depending on whether we need to do checksumming or not.
if (!sk->protinfo.af_inet.hdrincl) {
    <fill in IP header details>
    err = getfrag(frag, ((char *)iph)+iph->ihl*4, 0, length-iph->ihl*4);
}
else
    err = getfrag(frag, (void *)iph, 0, length);

Ok, so let’s take net/ipv4/udp.c::udp_getfrag_nosum. The code we care about is:
static int udp_getfrag_nosum(const void *p, char *to, unsigned int offset,
unsigned int fraglen)
{
    struct udpfakehdr *ufh = (struct udpfakehdr *)p;

    Copy the header part of the fake header
    memcpy(to, ufh, sizeof(struct udphdr));
    Now copy the data from the iovec into our buffer. See net/core/iovec.c::memcpy_fromiovecend
    return memcpy_fromiovecend(to+sizeof(struct udphdr), ufh->iov, offset,
    fraglen-sizeof(struct udphdr));
}

Jumping back to net/ipv4/ip_output.c::ip_build_xmit, we see the following. This does network filtering, then jumps to the
routine named as the last parameter.
err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL, rt->u.dst.dev,
output_maybe_reroute);
Then we come here to send the packet.

```c
output_maybe_reroute(struct sk_buff *skb) {
    return skb->dst->output(skb);
}
```