Proceedings of FREENIX Track:
2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

MBUF ISSUES IN 4.4BSD IPV6/IPSEC SUPPORT: EXPERIENCES FROM KAME IPV6/IPSEC IMPLEMENTATION

Jun-ichiro itojun Hagino

Mbuf issues in 4.4BSD IPv6/IPsec support
— experiences from KAME IPv6/IPsec implemntation —

Jun-ichiro itojun Hagino

KAME Project
Research Laboratory, Internet Initiative Japan Inc.
`http://www.kame.net/`
itojun@iijlab.net

ABSTRACT

The 4.4BSD network stack has made certain assumptions regarding the packets it will handle. In particular, 4.4BSD assumes that (1) the total protocol header length is shorter than or equal to MHLEN, usually 100 bytes, and (2) there are a limited number of protocol headers on a packet. Neither of these assumptions hold any longer, due to the way IPv6/IPsec specifications are written.

We at the KAME project are implementing IPv6 and IPsec support code on top of 4.4BSD. To cope with the problems, we have introduced the following changes: (1) a new function called *m_pulldown*, which adjusts the mbuf chain with a minimal number of copies/allocations, and (2) a new calling sequence for parsing inbound packet headers. These changes allow us to manipulate incoming packets in a safer, more efficient, and more spec-conformant way. The technique described in this paper is integrated into the KAME IPv6/IPsec stack kit, and is freely available under BSD copyright. The KAME codebase is being merged into NetBSD, OpenBSD and FreeBSD. An integration into BSD/OS is planned.

1. 4.4BSD incompatibility with IPv6/IPsec packet processing

The 4.4BSD network code holds a packet in a chain of ‘‘mbuf’’ structures. Each mbuf structure has three flavors:

- □ non-cluster header mbuf, which holds MHLEN (100 bytes in a 32bit architecture installation of 4.4BSD),
- □ non-cluster data mbuf, which holds MLEN (104 bytes), and
- □ cluster mbuf which holds MCLBYTES (2048 bytes).

We can make a chain of mbuf structures as a linked list. Mbuf chains will efficiently hold variable-length packet data. Such chains also enable us to insert or remove some of the packet data from the chain without data copies.

When processing inbound packets, 4.4BSD uses a function called *m_pullup* to ease the manipulation of data content in the mbufs. It also uses a deep function call tree for inbound packet processing. While these two items work just fine for traditional IPv4 processing, they do not work as well with IPv6 and IPsec processing.

1.1. Restrictions in 4.4BSD m_pullup

For input packet processing, the 4.4BSD network stack uses the *m_pullup* function to ease parsing efforts by adjusting the data content in mbufs for placement onto the continuous memory region. *m_pullup* is defined as follows:

```
struct mbuf *
m_pullup(m, len)
     struct mbuf *m;
     int len;
```

m_pullup will ensure that the first *len* bytes in the packet are placed in the continuous memory region. After a call to *m_pullup*, the caller can safely access the the first *len* bytes of the packet, assuming that they are continuous. The caller can, for example, safely use pointer variables into the continuous region, as long as they point inside the *len* boundary.

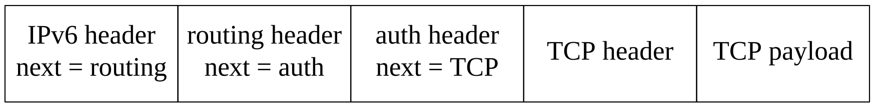

Figure 1: IPv6 extension header chain

m_pullup makes certain assumptions regarding protocol headers. *m_pullup* can only take *len* upto MHLEN. If the total packet header length is longer than MHLEN, *m_pullup* will fail, and the result will be a loss of the packet. Under IPv4 (Postel, 1981), the length assumption worked fine in most cases, since for almost every protocol, the total length of the protocol header part was less than MHLEN. Each packet has only two protocol headers, including the IPv4 header. For example, the total length of the protocol header part of a TCP packet (up to TCP data payload) is a maximum of 120 bytes. Typically, this length is 40 to 48 bytes. When an IPv4 option is present, it is stripped off before TCP header processing, and the maximum length passed to *m_pullup* will be 100.

1 The IPv4 header occupies 20 bytes.

2 The IPv4 option occupies 40 bytes maximum. It will be stripped off before we parse the TCP header. Also note that the use of IPv4 options is very rare.

3 The TCP header length is 20 bytes.

4 The TCP option is 40 bytes maximum. In most cases it is 0 to 8 bytes.

IPv6 specification (Deering, 1998) and IPsec specification (Kent, 1998) allow more flexible use of protocol headers by introducing chained extension headers. With chained extension headers, each header has a ‘‘next header field’’ in it. A chain of headers can be made as shown in Figure 2. The type of protocol header is determined by inspecting the previous protocol header. There is no restriction in the number of extension headers in the spec.

Because of extension header chains, there is now no upper limit in protocol packet header length. The *m_pullup* function would impose unnecessary restriction to the extension header processing. In addition, with the introduction of IPsec, it is now impossible to strip off extension headers during inbound packet processing. All of the data on the packet must be retained if it is to be authenticated using Authentication Header (Kent, 1998). Continuing the use of *m_pullup* will limit the number of extension headers allowed on the packet, and could jeopadize the possible usefulness of IPv6 extension headers. [1]

Another problem related to *m_pullup* is that it tends to copy the protocol header even when it is unnecessary to do so. For example, consider the mbuf chain shown in Figure 2:

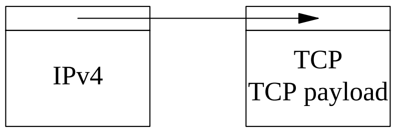

Figure 2: mbuf chain before *m_pullup*

Here, the first mbuf contains an IPv4 header in the continuous region, and the second mbuf contains a TCP header in the continuous region. When we look at the content of the TCP header, under 4.4BSD the code will look like the following:

```
struct ip *ip;
struct tcphdr *th;
ip = mtod(m, struct ip *);
/* extra copy with m_pullup */
m = m_pullup(m, iphdrlen + tcphdrlen);
/* MUST  reinit ip */
ip = mtod(m, struct ip *);
th = mtod(m, caddr_t) + iphdrlen;
```

As a result, we will get a mbuf chain shown in Figure 3.

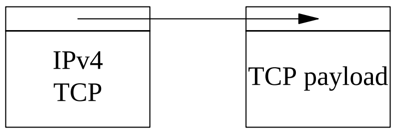

Figure 3: mbuf chain in figure 2 after *m_pullup*

Because *m_pullup* is only able to make a continuous region starting from the top of the mbuf chain, it copies the TCP portion in second mbuf into the first mbuf. The copy could be avoided if *m_pullup* were clever enough to handle this case. Also, the caller side is required to reinitialize all of the pointers that point to the content of mbuf, since after *m_pullup*, the first mbuf on the chain

[1] In IPv4 days, the IPv4 options turned out to be unusable due to a lack of implementation. This was because most commercial products simply did not support IPv4 options.

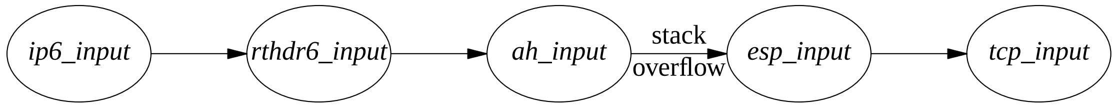

Figure 5: an excessively deep call chain can cause kernel stack overflow

can be reallocated and lives at a different address than before. While *m_pullup* design has provided simplicity in packet parsing, it is disadvantageous for protocols like IPv6.

The problems can be summarized as follows: (1) *m_pullup* imposes too strong restriction on the total length of the packet header (MHLEN); (2) *m_pullup* makes an extra copy even when this can be avoided; and (3) *m_pullup* requires the caller to reinitialize all of the pointers into the mbuf chain.

1.2. Protocol header processing with a deep function call chain

Under 4.4BSD, protocol header processing will make a chain of function calls. For example, if we have an IPv4 TCP packet, the following function call chain will be made (see Figure 4):

(1) *ipintr* will be called from the network software interrupt logic,

(2) *ipintr* processes the IPv4 header, then calls *tcp_input.*

(3) *tcp_input* will process the TCP header and pass the data payload to the socket queues.

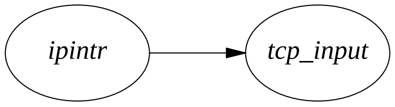

Figure 4: function call chain in IPv4 inbound packet processing

If chained extension headers are handled as described above, the kernel stack can overflow by a deep function call chain, as shown in Figure 5. IPv6/IPsec specifications do not define any upper limit to the number of extension headers on a packet, so a malicious party can transmit a ‘‘legal’’ packet with a large number of chained headers in order to attack IPv6/IPsec implementations. We have experienced kernel stack overflow in IPsec code, tunnelled packet processing code, and in several other cases. The IPsec processing routines tend to use a large chunk of memory on the kernel stack, in order to hold intermediate data and the secret keys used for encryption. [2] We cannot put the intermediate data region into a static data region outside of the kernel stack, because it would become a source of performance drawback on multiprocessors due to data locking.

Even though the IPv6 specifications do not define any restrictions on the number of extension headers, it may be possible to impose additional restriction in an IPv6 implementation for safety. In any case, it is not possible to estimate the amount of the kernel stack, which will be used by protocol handlers. We need a better calling convention for IPv6/IPsec header processing, regardless of the limits in the number of extension headers we may impose.

2. KAME approach

This section describes the approaches we at the KAME project took against the problems mentioned in the previous section. We introduce a new function called *m_pulldown,* in place of *m_pullup,* for adjusting payload data in the mbuf. We also change the calling sequence for the protocol input function.

2.1. What is the KAME project?

In the early days of IPv6/IPsec development, the Japanese research community felt it very important to make a reference code available in a freely-redistributable form for educational, research and deployment purposes. The KAME project is a consortium of 7 Japanese companies and an academic research group. The project aims to deliver IPv6/IPsec reference implementation for 4.4BSD, under BSD license. The KAME project intends to deliver the most spec-conformant IPv6/IPsec implementation possible.

2.2. m_pulldown function

Here we introduce a new function, *m_pulldown,* to address the 3 problems with *m_pullup* that we have described above. The actual source code is included at the end of this paper. The function prototype is as follows:

[2] For example, blowfish encryption processing code typically uses an intermediate data region of 4K or more. With typical 4.4BSD installation on i386 architecture, the kernel stack region occupies less than 8K bytes and does not grow on demand.

```
struct mbuf *
m_pulldown(m, off, len, offp)
     struct mbuf *m;
     int off, len;
     int *offp;
```

m_pulldown will ensure that the data region in the mbuf chain, starting at *off* and ending at *off* + *len*, is put into a continuous memory region. *len* must be smaller than, or equal to, MCLBYTES (2048 bytes). The function returns a pointer to an intermediate mbuf in the chain (we refer to the pointer as *n*), and puts the new offset in *n* to **offp*. If *offp* is NULL, the resulting region can be located by *mtod(n, caddr_t)*; if *offp* is non-null, it will be located at *mtod(n, caddr_t)* + **offp*. The mbuf prior to *off* will remain untouched, so it is safe to keep the pointers to the mbuf chain. For example, consider the mbuf chain on Figure 6 as the input.

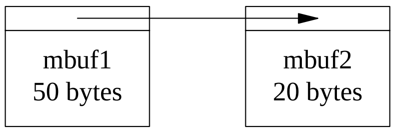

Figure 6: mbuf chain before the call to *m_pulldown*

If we call *m_pulldown* with *off = 40*, *len = 10*, and a non-null *offp*, the mbuf chain will remain unchanged. The return value will be a pointer to mbuf1, and **offp* will be filled with 40. If we call *m_pulldown* with *off = 40*, *len = 20*, and null *offp*, then the mbuf chain will be modified as shown in Figure 7, by allocating a new mbuf, mbuf3, into the middle and moving data from both mbuf1 and mbuf2. The function returns a pointer to mbuf3.

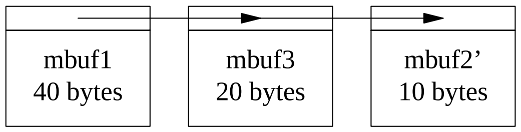

Figure 7: mbuf chain after call to *m_pulldown*, with *off = 40* and *len = 20*

The *m_pulldown* function solves all 3 problems in *m_pullup* that were described in the previous section. *m_pulldown* does not copy mbufs when copying is not necessary. Since it does not modify the mbuf chain prior to the speficied offset *off*, it is not necessary for the caller to re-initialize the pointers into the mbuf data region. With *m_pullup*, we always needed to specify the data payload length, starting from the very first byte in the packet. With *m_pulldown*, we pass *off* as the offset to the data payload we are interested in. This change avoids extra data manipulation when we are only interested in the intermediate data portion of the packet. It also eases the assumption regarding total packet header length. While *m_pullup* assumes that the total packet header length is smaller than or equal to MHLEN (100 bytes), *m_pulldown* assumes that single packet header length is smaller than or equal to MCLBYTES (2048 bytes). With mbuf framework this is the best we can do, since there is no way to hold continuous region longer than MCLBYTES in a standard mbuf chain.

2.3. New function prototype for inbound packet processing

For IPv6 processing, our code does not make a deep function call chain. Rather, we make a loop in the very last part of *ip6_input*, as shown in Figure 8. IPPROTO_DONE is a pseudo-protocol type value that identifies the end of the extension header chain. If more protocol headers exist, each header processing code will update the pointer variables and return the next extension header type. If the final header in the chain has been reached, IPPROTO_DONE is returned. With this code, we no longer have a deep call chain for IPv6/IPsec processing. Rather, *ip6_input* will make calls to each extension header processor directly. This avoids the possibility of overflowing the kernel stack due to multiple extension header processing.

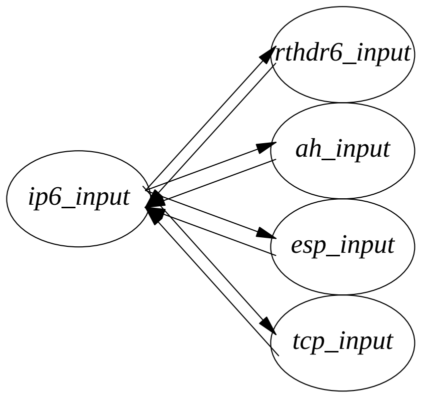

Figure 9: KAME avoids function call chain by making a loop in *ip6_input*

Regardless of the calling sequence imposed by the *pr_input* function prototype, it is important not to use up the kernel stack region in protocol handlers. Sometimes it is necessary to decrease the size of kernel stack usage by using pointer variables and dynamically allocated regions.

```
struct ip6protosw {
     int (*pr_input) __P((struct mbuf **, int *, int));
     /* and other members */
};

ip6_input(m)
     struct mbuf *m;
{
     /* in the very last part */
     extern struct ip6protosw inet6sw[];
     /* the first one in extension header chain */
     nxt = ip6.ip6_nxt;
     while (nxt != IPPROTO_DONE)
          nxt = (*inet6sw[ip6_protox[nxt]].pr_input)(&m, &off, nxt);
}

/* in each header processing code */
int
foohdr_input(mp, offp, proto)
     struct mbuf **mp;
     int *offp;
     int proto;
{
     /* some processing, may modify mbuf chain */

     if (we have more header to go) {
          *mp = newm;
          *offp = nxtoff;
          return nxt;
     } else {
          m_freem(newm);
          return IPPROTO_DONE;
     }
}
```

Figure 8: KAME IPv6 header chain processing code.

3. Alternative approaches

Many BSD-based IPv6 stacks have been implemented. While the most popular stacks include NRL, INRIA and KAME, dozens of other BSD-based IPv6 implementations have been made. This section presents alternative approaches for purposes of comparison.

3.1. NRL m_pullup2

The latest NRL IPv6 release copes with the *m_pullup* limitation by introducing a new function, *m_pullup2*. *m_pullup2* works similarly to *m_pullup*, but it allows *len* to extend up to MCLBYTES, which corresponds to 2048 bytes in a typical installation. When the *len* parameter is smaller than or equal to MHLEN, *m_pullup2* simply calls *m_pullup* from the inside.

While *m_pullup2* works well for packet headers up to MCLBYTES with very little change in code, it does not avoid making unnecessary copies. It also imposes restrictions on the total length of packet headers. The assumption here is that the total length of packet headers is less than MCLBYTES.

3.2. Hydrangea changes to m_devget

The Hydrangea IPv6 stack was implemented by a group of Japanese researchers, and is one of the ancestors of the KAME IPv6 stack. The Hydrangea IPv6 stack avoids the need for *m_pullup* by modifying the mbuf allocation policy in drivers. For inbound packets, the drivers allocate mbufs by using the *m_devget* function, or by re-implementing the behavior of *m_devget*. *m_devget* allocates mbuf as follows:

1 If the packet fits in MHLEN (100 bytes), allocate a single non-cluster mbuf.

2 If the packet is larger than MHLEN but fits in MHLEN + MLEN (204 bytes), allocate two non-cluster mbufs.

3 Otherwise, allocate multiple cluster mbufs, MCLBYTES (2048 bytes) in size.

For typical packets, the second case is where *m_pullup* is used. The Hydrangea stack avoids the use of *m_pullup* by eliminating the second case.

This approach worked well in most cases, but failed for (1) loopback interface, (2) tunnelled packets, and (3) non-conforming drivers. With the Hydrangea approach, every device driver had to be examined to ensure the new mbuf allocation policy. We could not be sure if the constraint was guaranteed until we checked the driver code, and the Hydrangea approach raised many support issues. This was one of our motivations for introducing *m_pulldown.*

4. Comparisons

This section compares the following three approaches in terms of their characteristics and actual behavior: (1) 4.4BSD *m_pullup,* (2) NRL *m_pullup2,* and (3) KAME *m_pulldown.*

4.1. Comparison of assumption

Table 1 shows the assumptions made by each of the three approaches. As mentioned earlier, *m_pullup* imposes too stringent requirement for the total length of packet headers. *m_pullup2* is workable in most cases, although this approach adds more restrictions than the specification claims. *m_pulldown* assumes that the single packet header is smaller than MCLBYTES, but makes no restriction regarding the total length of packet headers. With a standard mbuf chain, this is the best *m_pulldown* can do, since there is no way to hold continuous region longer than MCLBYTES. This characteristic can contribute to better specification conformance, since *m_pulldown* will impose fewer additional restrictions due to the requirements of implementation.

Among the three approaches, only *m_pulldown* avoids making unnecessary copies of intermediate header data and avoids pointer reinitialization after calls to these functions. These attributes result in smaller overhead during input packet processing.

At present, we know of no other 4.4BSD-based IPv6/IPsec stack that addresses kernel stack overflow issues, although we are open to new perspectives and new information.

4.2. Performance comparison based on simulated statistics

To compare the behavior and performance of *m_pulldown* against *m_pullup* and *m_pullup2* using the same set of traffic and mbuf chains, we have gathered simulated statistics for *m_pullup* and *m_pullup2,* in *m_pulldown* function. By running a kernel using the modified *m_pulldown* function, we can easily gather statistics for these three functions against exactly the same traffic.

The comparison was made on a computer (with Celeron 366MHz CPU, 192M bytes of memory) running NetBSD 1.4.1 with the KAME IPv6/IPsec stack. Network drivers allocate mbufs just as normal 4.4BSD does. *m_pulldown* is called whenever it is needed to ensure continuity in packet data during inbound packet processing. The role of the computer is as an end node, not a router.

To describe the content of the following table, we must look at the source code fragment. Figure 10 shows the code fragment from our source code. The code fragment will (1) make the TCP header on the mbuf chain *m* at offset *hdrlen* continuous, and (2) point the region with pointer *th.* We use a macro named IP6_EXTHDR_CHECK, and the code before and after the macro expansion is shown in the figure.

```
/* ensure that *th from hdrlen is continuous */
/* before macro expansion... */
struct tcphdr *th;
IP6_EXTHDR_CHECK(th, struct tcphdr *, m,
      hdrlen, sizeof(*th));
if (th == NULL)
    return;   /*m is already freed*/


/* after macro expansion... */
struct tcphdr *th;
int off;
struct mbuf *n;
if (m->m_len < hdrlen + sizeof(*th)) {
    n = m_pulldown(m, hdrlen, sizeof(*th), &off);
    if (n)
       th = (struct tcphdr *)(mtod(n, caddr_t) + off);
    else
       th = NULL;
} else
    th = (struct tcphdr *)(mtod(m, caddr_t) + hdrlen);
if (th == NULL)
    return;
```

Figure 10: code fragment for trimming mbuf chain.

In Table 2, the first column identifies the test case. The second column shows the number of times the IP6_EXTHDR_CHECK macro was used. In other words, it shows the number of times we have made checks against mbuf length. The remaining columns show, from left to right, the number of times memory allocation/copy was performed in each of the variants. In the case of *m_pullup,* we counted the number of cases we passed *len* in excess of MHLEN (96 bytes in this installation). This result suggests that there was no packet with a packet header portion larger than MCLBYTES (2048 bytes). In the evaluation we have used *m_pulldown* against IPv6 traffic only.

	m_pullup	*m_pullup2*	*m_pulldown*
total header length single header length	MHLEN(100) –	MCLBYTES(2048) –	– MCLBYTES(2048)
avoids copy on intermediate headers	no	no	yes
avoids pointer reinitialization	no	no	yes

Table 1: assumptions in mbuf manipulation approaches.

test	len checks	*m_pulldown*			*m_pullup*			*m_pullup2*	
		call	alloc	copy	alloc	copy	fail	alloc	copy
(1)	204923	1706	1595	1596	165	165	1541	1596	1596
(2)	1063995	23786	22931	23008	1171	1229	22557	22895	22953
(3)	520028	1245	948	957	432	432	813	945	945
(4)	438602	180	6	6	178	178	2	24	24
(5)	5570	2236	206	206	812	812	1424	1424	1424

Table 2: number of mbuf allocation/copy against traffic

test	IPv6 input	TCP	UDP	ICMPv6	1 mbuf	2 mbufs	ext mbuf(s)
(1)	29334	20892	2699	5739	3624	15632	10078
(2)	313218	215919	15930	80263	38751	172976	101491
(3)	132267	117822	8561	5882	12782	59799	59686
(4)	73160	66512	5249	1343	7475	42053	23632
(5)	1433	148	53	52	103	1203	127

Table 3: Traffic characteristics for tests in Table 2

From these measured results, we obtain several interesting observations. *m_pullup* actually failed on IPv6 trafic. If an IPv6 implementation uses *m_pullup* for IPv6 input processing, it must be coded carefully so as to avoid trying *m_pullup* against any length longer than MHLEN. To achieve this end, the code copies the data portion from the mbuf chain to a separate buffer, and the cost of memory copies becomes a penalty.

Due to the nature of this simulation, the comparison described above may contain an implicit bias. Since the IPv6 protocol processing code is written by using *m_pulldown*, the code is somewhat biased toward *m_pulldown*. If a programmer had to write the entire IPv6 protocol processing with *m_pullup* only, he or she would use *m_copydata* to copy intermediate extension headers buried deep inside the header chains, thus making it unnecessary to call *m_pullup*. In any case, a call to *m_copydata* will result in a data copy, which causes extra overhead.

In all cases, the number of length checks (second column) exceeds the number of inbound packets. This behavior is the same as in the original 4.4BSD stack; we did not add a significant number of length checks to the code. This is because *m_pulldown* (or *m_pullup* in the 4.4BSD case) is called as necessary during the parsing of the headers. For example, to process a TCP-over-IPv6 packet, at least 3 checks would be made against m->m_len; these checks would be made to grab the IPv6 header (40 bytes), to grab the TCP header (20 bytes), and to grab the TCP header and options (20 to 60 bytes). The length of the TCP option part is kept inside the TCP header, so the length needs to be checked twice for the TCP part.

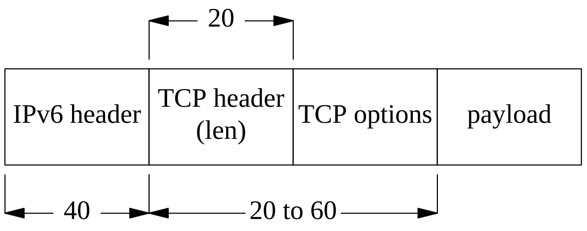

Figure 11: processing a TCP-over-IPv6 packet requires 3 length checks.

The results suggest that we call *m_pulldown* more frequently in ICMPv6 processing than in the processing of other protocols. These additional calls are made for parsing of ICMPv6 and for neighbor discovery options. The use of loopback interface also contributes to the use of *m_pulldown*.

In the tests, the number of copies made in the *m_pullup2* case is similar to the number made in the *m_pulldown* case. *m_pulldown* makes less copies than *m_pullup2* against packets like below:

- □ A packet is kept in multiple mbuf. With mbuf allocation policy in *m_devget*, we will see two mbufs to hold single packet if the packet is larger than MHLEN and smaller than MHLEN + MLEN, or the packet is larger than MCLBYTES.
- □ We have extension headers in multiple mbufs. Header portion in the packet needs to occupy first mbuf and subsequent mbufs.

To demonstrate the difference, we have generated an IPv6 packet with a routing header, with 4 IPv6 addresses. The test result is presented as the 5th test in Table 2. Packet will look like Figure 12. First 112 bytes are occupied by an IPv6 header and a routing header, and the remaining 16 bytes are used for an ICMPv6 header and payload. The packet met the above condition, and *m_pulldown* made less copies than *m_pullup2*. To process single incoming ICMPv6 packet shown in the figure, *m_pullup2* made 7 copies while *m_pulldown* made only 1 copy.

```
node A (source) = 2001:240:0:200:260:97ff:fe07:69ea
node B (destination) = 2001:240:0:200:a00:5aff:fe38:6f86
17:39:43.346078 A > B:
       srcrt (type=0,segleft=4,[0]B,[1]B,[2]B,[3]B):
       icmp6: echo request (len 88, hlim 64)
               6000 0000 0058 2b40 2001 0240 0000 0200
               0260 97ff fe07 69ea 2001 0240 0000 0200
               0a00 5aff fe38 6f86 3a08 0004 0000 0000
               2001 0240 0000 0200 0a00 5aff fe38 6f86
               2001 0240 0000 0200 0a00 5aff fe38 6f86
               2001 0240 0000 0200 0a00 5aff fe38 6f86
               2001 0240 0000 0200 0a00 5aff fe38 6f86
               8000 b650 030e 00c8 ce6e fd38 d553 0700
```

Figure 12: Packets with IPv6 routing header.

During the test, we experienced no kernel stack overflow, thanks to a new calling sequence between IPv6 protocol handlers.

The number of copies and mbuf allocations vary very much by tests. We need to investigate the traffic characteristic more carefully, for example, about the average length of header portion in packets.

5. Related work

Van Jacobson proposed pbuf structure [3] as an alternative to BSD mbuf structure. The proposal has two main arguments. First is the use of continuous data buffer, instead of chained fragments like mbufs. Another is the improvement to TCP performance by restructuring TCP input/output handling. While the latter point still holds for IPv6, we believe that the former point must be reviewed carefully before being used with IPv6. Our experience suggests that we need to insert many intermediate extension headers into the packet data during IPv6 outbound packet processing. We believe that mbuf is more suitable than the proposed pbuf structure for handling the packet data efficiently. Using pbuf may result in the making of more copies than in the mbuf case.

In a cross-BSD portability paper (Metz, 1999), Craig Metz described *nbuf* structure in NRL IPv6/IPsec stack. nbuf is a wrapper structure used to unify linux linear-buffer packet management and BSD mbuf structure, and is not closely related to the topic of this paper. The *m_pullup2* example discussed in this paper is drawn from the NRL implementation.

6. Conclusions

This paper discussed mbuf manipulation in a 4.4BSD-based IPv6/IPsec stack, namely KAME IPv6/IPsec implementation. 4.4BSD makes certain assumptions regarding packet header length and its format. For IPv6/IPsec support, we removed those assumptions from the 4.4BSD code. We introduced the *m_pulldown* function and a new function call sequence for inbound packet processing. These innovations helped us to implement IPv6/IPsec in a very spec-conformant manner, with fewer implementation restrictions added against specifications.

The described code is publically available, under a BSD-like license, at `ftp://ftp.kame.net/`. KAME IPv6/IPsec stack is being merged into 4.4BSD variants like FreeBSD, NetBSD and OpenBSD. An integration into BSD/OS is planned. We will be able to see official releases of these OSes with KAME code soon.

7. Acknowledgements

The paper was made possible by the collective efforts of researchers at the KAME project and the WIDE project and of other IPv6 implementers at large. We would also like to acknowledge all four BSD groups who helped us improve the KAME IPv6 stack code by sending bug reports and improvement suggestions, and the Freenix reviewers helped polish the paper.

[3] A reference should be here, but I'm having hard time finding published literature for it.

References

Postel, 1981.
John Postel, “Internet Protocol” in *RFC791* (September 1981). ftp://ftp.isi.edu/in-notes/rfc791.txt.

Deering, 1998.
S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification” in *RFC2460* (December 1998). ftp://ftp.isi.edu/in-notes/rfc2460.txt.

Kent, 1998.
Stephen Kent and Randall Atkinson, “Security Architecture for the Internet Protocol” in *RFC2401* (November 1998). ftp://ftp.isi.edu/in-notes/rfc2401.txt.

Kent, 1998.
Stephen Kent and Randall Atkinson, “IP Authentication Header” in *RFC2402* (November 1998). ftp://ftp.isi.edu/in-notes/rfc2402.txt.

Metz, 1999.
Craig Metz, “Porting Kernel Code to Four BSDs and Linux” in *1999 USENIX annual technical conference, Freenix track* (June 1999). http://www.usenix.org/publications/library/proceedings/usenix99/metz.html.

```
/*
 * ensure that [off, off + len) is contiguous on the mbuf chain "m".
 * packet chain before "off" is kept untouched.
 * if offp == NULL, the target will start at <retval, 0> on resulting chain.
 * if offp != NULL, the target will start at <retval, *offp> on resulting chain.
 *
 * on error return (NULL return value), original "m" will be freed.
 *
 * XXX M_TRAILINGSPACE/M_LEADINGSPACE on shared cluster (sharedcluster)
 */
struct mbuf *
m_pulldown(m, off, len, offp)
        struct mbuf *m;
        int off, len;
        int *offp;
{
        struct mbuf *n, *o;
        int hlen, tlen, olen;
        int sharedcluster;

        /* check invalid arguments. */
        if (m == NULL)
                panic("m == NULL in m_pulldown()");
        if (len > MCLBYTES) {
                m_freem(m);
                return NULL;    /* impossible */
        }

        n = m;
        while (n != NULL && off > 0) {
                if (n->m_len > off)
                        break;
                off -= n->m_len;
                n = n->m_next;
        }
        /* be sure to point non-empty mbuf */
        while (n != NULL && n->m_len == 0)
                n = n->m_next;
        if (!n) {
                m_freem(m);
                return NULL;    /* mbuf chain too short */
        }

        /*
         * the target data is on <n, off>.
         * if we got enough data on the mbuf "n", we're done.
         */
        if ((off == 0 || offp) && len <= n->m_len - off)
                goto ok;

        /*
         * when len < n->m_len - off and off != 0, it is a special case.
         * len bytes from <n, off> sits in single mbuf, but the caller does
         * not like the starting position (off).
         * chop the current mbuf into two pieces, set off to 0.
         */
        if (len < n->m_len - off) {
                o = m_copym(n, off, n->m_len - off, M_DONTWAIT);
                if (o == NULL) {
                        m_freem(m);
                        return NULL;    /* ENOBUFS */
                }
                n->m_len = off;
                o->m_next = n->m_next;
                n->m_next = o;
                n = n->m_next;
                off = 0;
                goto ok;
        }

        /*
         * we need to take hlen from <n, off> and tlen from <n->m_next, 0>,
         * and construct contiguous mbuf with m_len == len.
         * note that hlen + tlen == len, and tlen > 0.
         */
        hlen = n->m_len - off;
        tlen = len - hlen;

        /*
         * ensure that we have enough trailing data on mbuf chain.
         * if not, we can do nothing about the chain.
         */
        olen = 0;
        for (o = n->m_next; o != NULL; o = o->m_next)
                olen += o->m_len;
        if (hlen + olen < len) {
                m_freem(m);
                return NULL;    /* mbuf chain too short */
        }

        /*
         * easy cases first.
         * we need to use m_copydata() to get data from <n->m_next, 0>.
         */
        if ((n->m_flags & M_EXT) == 0)
                sharedcluster = 0;
        else {
                if (n->m_ext.ext_free)
                        sharedcluster = 1;
                else if (MCLISREFERENCED(n))
                        sharedcluster = 1;
                else
                        sharedcluster = 0;
        }
        if ((off == 0 || offp) && M_TRAILINGSPACE(n) >= tlen
         && !sharedcluster) {
                m_copydata(n->m_next, 0, tlen, mtod(n, caddr_t) + n->m_len);
                n->m_len += tlen;
                m_adj(n->m_next, tlen);
                goto ok;
        }
        if ((off == 0 || offp) && M_LEADINGSPACE(n->m_next) >= hlen
         && !sharedcluster) {
                n->m_next->m_data -= hlen;
                n->m_next->m_len += hlen;
                bcopy(mtod(n, caddr_t) + off, mtod(n->m_next, caddr_t), hlen);
                n->m_len -= hlen;
                n = n->m_next;
                off = 0;
                goto ok;
        }

        /*
         * now, we need to do the hard way.  don't m_copy as there's no room
         * on both end.
         */
        MGET(o, M_DONTWAIT, m->m_type);
        if (o == NULL) {
                m_freem(m);
                return NULL;    /* ENOBUFS */
        }
        if (len > MHLEN) {      /* use MHLEN just for safety */
                MCLGET(o, M_DONTWAIT);
                if ((o->m_flags & M_EXT) == 0) {
                        m_freem(m);
                        m_free(o);
                        return NULL;    /* ENOBUFS */
                }
        }
        /* get hlen from <n, off> into <o, 0> */
        o->m_len = hlen;
        bcopy(mtod(n, caddr_t) + off, mtod(o, caddr_t), hlen);
        n->m_len -= hlen;
        /* get tlen from <n->m_next, 0> into <o, hlen> */
        m_copydata(n->m_next, 0, tlen, mtod(o, caddr_t) + o->m_len);
        o->m_len += tlen;
        m_adj(n->m_next, tlen);
        o->m_next = n->m_next;
        n->m_next = o;
        n = o;
        off = 0;

ok:
        if (offp)
                *offp = off;
        return n;
}
```