Geoff Huston
October 2017

Raw Sockets in IPv6

Among many other functions performed by a computer's operating system, there is typically an interface to a shared local network protocol engine. This means that applications that run within the operating system's environment don't need to implement their own network protocol engine, as they can make use of a shared common interface to the underlying network protocol engine via a simple standard interface. In Unix, the commonly used API to the underlying communications system is via the *socket* routines. Sockets creates an abstract model of the underlying network by mimicking a simple peripheral device, and once the application connects to a network socket (akin to opening a virtual connection to a remote host), it can then use the network through conventional *read* and *write* commands on that socket. Using the appropriate socket abstraction, the details about managing the TCP or UDP packet headers and the common IP layer is managed by the operating system's protocol drivers, and is largely hidden from the application.

As part of a measurement experiment, we wanted an implementation of an IPv6 UDP server and a TCP server that generated fragmented IPv6 packets. However, as an added condition, we wanted the application to directly control the packet fragmentation function. The conventional standard socket interface masks any visibility to the underlying packet transactions, and therefore cannot be used for this experiment. There is a specialised socket option that allows an application to interact directly with the underlying communications driver and read and write IP datagrams without having the packets processed by the operating system's IP protocol drivers. This is the *Raw Socket* option in IPv4, and an example of opening such a socket is shown in this C code snippet:

```
/*
 * open_raw_socket
 *
 * open a raw socket interface into the kernel
 */

void
open_raw_socket()
  {
  const int on = 1 ;

  /* create the raw socket via the socket call*/
  if ((sock_fd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("socket() error");
    exit(EXIT_FAILURE);
    }

  /* inform the kernel the IP header is already attached via a socket option */
  if (setsockopt(sock_fd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    perror("setsockopt() error");
    exit(EXIT_FAILURE);
    }
  }
```

This code snippet is written to use IPv4. A shift to IPv6 does not entail a lot of code changes. It's a case of changing the protocol specifier from `AF_INET` to `AF_INET6`, expanding the size of the IP address data structures to 128 bits and there is not that much else to do. But this is not the case for raw sockets. Here IPv6 is indeed very different, as noted in RFC 3542:

> [The] difference from IPv4 raw sockets is that complete packets (that is, IPv6 packets with extension headers) cannot be sent or received using the IPv6 raw sockets API.
> …
> When writing to a raw socket the kernel will automatically fragment the packet if its size exceeds the path MTU, inserting the required fragment headers.
> …
> Most IPv4 implementations give special treatment to a raw socket created with a third argument to socket() of IPPROTO_RAW, whose value is normally 255, to have it mean that the application will send down complete packets including the IPv4 header. (Note: This feature was added to IPv4 in 1988 by Van Jacobson to support traceroute, allowing a complete IP header to be passed by the application, before the IP_HDRINCL socket option was added.) We note that IPPROTO_RAW has no special meaning to an IPv6 raw socket (and the IANA currently reserves the value of 255 when used as a next-header field).
>
> RFC 3542, "Advanced Sockets Application Program Interface (API) for IPv6", W. Stevens et al, May 2003.

What we want to do in this measurement experiment, namely use an IPv6 raw socket that allows the application to have direct control of the entirety of the IPv6 packet header, including fragmentation handling, is not directly supported using the IPv6 raw IP socket interface.

But there is another approach that can be used. Sockets can extend one further level down in the protocol stack, and connect directly to the network interface rather than to a network protocol engine. This form of raw socket requires the application to write complete packets to the socket interface, including the media layer framing (such as the Ethernet headers), the IP level packet header, the transport protocol header, as well as any payload. If we want to perform explicit control over the generation of IPv6 Extension Headers to support IPv6 packet fragmentation control at the application level, then this media level socket interface looks like a viable approach.

In this article, I will describe how we used this raw socket interface in IPv6 to generate a UDP-based DNS server and a TCP-based HTTP(S) server that allowed the application to exercise direct control over packet fragmentation.

A Raw Socket UDP DNS Server

The aim of this packet handler was to provide a font end to a conventional DNS server. Incoming IPv6 UDP DNS queries are passed to a conventional "back end" DNS resolver. UDP responses from the "back end" DNS resolver are fragmented into at least two IPv6 packets, ensuring that no packet is larger than 512 octets, and passed back to the original sender.

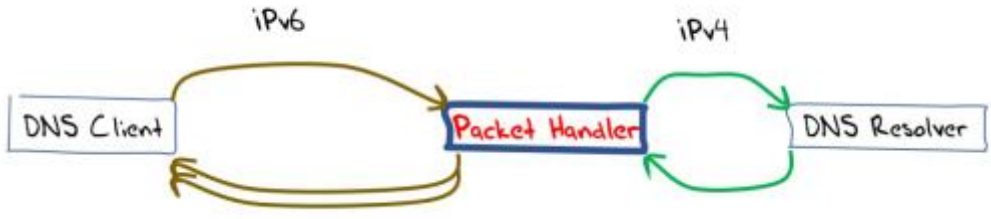

Figure 1 – UDP Packet Handler

Let's now look at the details of the code for this packet handler. The first part of the packet processor is a conventional UDP server listening port that listens on IPv6 to incoming packets addressed to the local port 53.

```
/* Create a IPv6 datagram socket and associate it with the variable sockfd*/
sockfd = socket(AF_INET6,SOCK_DGRAM,17);
if (sockfd < 0) {
  perror("Socket");
  exit(EXIT_FAILURE);
  }

/* The local IPv6 address used to listen is stored in the variable host
   This is converted to an internal representation of the IPv6 address */
if (((status = inet_pton(AF_INET6,host,&listen.sin6_addr))) <= 0) {
  if (!status)
    fprintf(stderr, "Not in presentation format");
  else
    perror("inet_pton");
  exit(EXIT_FAILURE);
  }

/* set this address into the socket structure and use port 53 (DNS) */
listen.sin6_family = AF_INET6 ;
listen.sin6_port = htons(53);

/* now bind the socket to the local IPv6 address and port 53 */
if (bind(sockfd,(const struct sockaddr *) &listen,(sizeof listen))) {
  perror("bind") ;
  exit(EXIT_FAILURE) ;
  }

/* we can now set up the listener - the source address and source port of
   the incoming UDP packet is stored in the cliaddr struct  */
addrlen= sizeof *cliaddr ;
cliaddr=malloc(addrlen);
len=addrlen;

for ( ;; ) {     /* do forever */

  if ((rc = recvfrom(sockfd, buf, MAXBUF, 0, (struct sockaddr *) cliaddr, &len)) < 0 ) {
    printf("server error: errno %d\n",errno);
    perror("reading datagram");
    exit(1);
    }

  /* at this stage the client's address and port is stored in cliaddr, and the DNS query
     is stored in buf, with length rc - we can pass this DNS query to the back end */
```

The next part of the code is also quite conventional. It takes the original DNS query, which is the payload returned by the `recvfrom()` call, and passes it to a DNS resolver without alteration. This example code is synchronous for simplicity, so the routine to query the back end DNS resolver will wait for the DNS resolver's response before returning. The code also uses IPv4 for the connection to the DNS server.

```
/* take an incoming UDP DNS query and flick it to a back-end DNS processor
   and collect the UDP response */

/* open a UDP socket to the DNS server at address serveraddr4 */

if ((dns_sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  perror("UDP slave socket() error") ;
  return 1;
  }
if (connect(dns_sockfd, (struct sockaddr *) &serveraddr4, sizeof(serveraddr4)) < 0) {
  perror("UDP V4 connect error") ;
  return 1;
  }

/* send the DNS query to the server */
i = write(sockfd, query, data_size) ;
if (i < 0) {
  perror("ERROR writing to socket");
  return 1;
  }

/* collect the server's reply */
dns_response.len = recvfrom(dns_sockfd, dns_response.buf, MAXBUF, 0,
                            (struct sockaddr *) &serveraddr4, &serverlen);
if (dns_response.len < 0) {
  perror("UDP recvfrom");
  return 1;
  }
close(dns_sockfd) ;
```

This is the simplest approach, but obviously not the most efficient. This code could be made asynchronous by using a non-blocking read calls on the socket, but for the sake of simplicity we will use the synchronous call to the socket.

The last part of the packet handler is the processing of IPv6 outbound UDP packets directed back to the original client, we here we will use a raw Ethernet socket interface.

The first part of this code opens a raw socket to in the interface named by the variable `interface`. In order to construct Ethernet packets, we need to get the Ethernet MAC address of the interface, which is performed by an `ioctl` call on a raw socket interface. We also use the `if_nametoindex()` call to get the local index value of the named interface. We can then open a raw Ethernet socket on the named interface.

```
/* Get a socket descriptor to look up interface */
if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
  perror("socket() failed to get socket descriptor for using ioctl() ");
  exit(EXIT_FAILURE);
  }

/* Use an ioctl() to look up interface name and get its MAC address */
/* clear the ifr variable */
memset(&ifr, 0, sizeof (ifr));

/* write in the name of the interface: held in variable interface */
snprintf(ifr.ifr_name, sizeof (ifr.ifr_name), "%s", interface);
if (ioctl(sd, SIOCGIFHWADDR, &ifr) < 0) {
  perror("ioctl() failed to get source MAC address ");
  exit (EXIT_FAILURE);
  }
/* done! */
close(sd);

/* Copy source MAC address into src_mac */
memcpy(src_mac, ifr.ifr_hwaddr.sa_data, 6 * sizeof (uint8_t));

/* Find interface index from interface name using the call if_nametoindex()
   and store the index value in struct sockaddr_ll device (which will be
   used as an argument of sendto() of the subsequent output call */
memset(&device, 0, sizeof (device));
if ((device.sll_ifindex = if_nametoindex(interface)) == 0) {
  perror ("if_nametoindex() failed to obtain interface index ");
  exit (EXIT_FAILURE);
  }

/* now copy over the mac address */
device.sll_family = AF_PACKET;
memcpy(device.sll_addr, src_mac, 6 * sizeof (uint8_t));
device.sll_halen = 6;

/* now we are ready to submit a request for a raw socket descriptor */
if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
  perror ("socket() failed ");
  exit (EXIT_FAILURE);
  }
```

There is still one critical piece of information that is missing here. When we generate the Ethernet packet, we need to add the destination MAC address of the local IPv6 gateway. One way is to use the `ifconfig` cli command and pull this information manually, assuming of course that such a command exists on your platform. We can also try to perform this automatically by listening for periodic IPv6 router advertisements on the interface. The MAC source address of these router advertisements packets is the destination address we are after. Here's a code snippet that binds to the interface and looks for RA messages:

```
/* Request a socket descriptor sd */
if ((sd = socket (AF_INET6, SOCK_RAW, IPPROTO_ICMPV6)) < 0) {
  perror ("Failed to get socket descriptor ");
  exit (EXIT_FAILURE);
  }
```

```
/* Set flag so we receive destination address from recvmsg */
on = 1;
if ((status = setsockopt (sd, IPPROTO_IPV6, IPV6_RECVPKTINFO, &on, sizeof (on))) < 0) {
  perror ("setsockopt to IPV6_RECVPKTINFO failed ");
  exit (EXIT_FAILURE);
  }

/* Obtain MAC address of this interface */
memset (&ifr, 0, sizeof (ifr));
snprintf (ifr.ifr_name, sizeof (ifr.ifr_name), "%s", interface);
if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {
  perror ("ioctl() failed to get source MAC address ");
  exit (EXIT_FAILURE);
  }

/* Retrieve interface index of this node */
if ((ifindex = if_nametoindex (interface)) == 0) {
  perror ("if_nametoindex() failed to obtain interface index ");
  exit (EXIT_FAILURE);
  }

/* Bind a device socket to this interface */
if (setsockopt (sd, SOL_SOCKET, SO_BINDTODEVICE, (void *) &ifr, sizeof (ifr)) < 0) {
  perror ("SO_BINDTODEVICE failed");
  exit (EXIT_FAILURE);
  }

/* Listen for incoming message from socket sd
   Keep at it until we get a router advertisement */
ra = (struct nd_router_advert *) inpack;
while (ra->nd_ra_hdr.icmp6_type != ND_ROUTER_ADVERT) {
  if ((len = recvmsg (sd, &msghdr, 0)) < 0) {
    perror ("recvmsg failed ");
    exit (EXIT_FAILURE);
    }
  }

pkt = (uint8_t *) inpack;

for (i=2; i<=7; i++) {
  dst_mac[i-2] = pkt[sizeof (struct nd_router_advert) + i];
  }
close (sd);
```

We start the raw socket write process by generating a IPv6 UDP packet.

```
/* IPv6 header */
iphdr = (struct ip6_hdr *) &out_packet_buffer[0] ;

/* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) */
iphdr->ip6_flow = htonl ((6 << 28) | (0 << 20) | 0);

/* Next header (8 bits): 44 for Frag */
iphdr->ip6_nxt = 44;

/* Hop limit (8 bits): default to maximum value */
iphdr->ip6_hops = 255;

/* src address */
bcopy(&srcaddr->sin6_addr,&(iphdr->ip6_src), 16) ;

/* dst address */
bcopy(&cliaddr->sin6_addr,&(iphdr->ip6_dst), 16);

/* set up the UDP packet */
uhdr = (struct udphdr *) &(payload[0]);
uhdr->uh_sport = htons(port) ;
uhdr->uh_dport = cliaddr->sin6_port;
uhdr->uh_ulen = htons(dns_response->len + 8);
uhdr->uh_sum = 0;

/* copy payload bytes from the dns response buffer to the payload buffer */
bcopy(dns_response->buf,&payload[8],dns_response->len) ;

/* calculate the UDP checksum */
uhdr->uh_sum = udp_checksum(uhdr,dns_response->len + 8, &srcaddr->sin6_addr, &cliaddr->sin6_addr);
```

We have deliberately kept the IPv6 packet header and the UDP header and DNS response payload in separate memory buffers. This will allow us to place an IPv6 Extension Header into each output packet between the IPv6 packet header and each component part of the fragmented UDP header and payload, if we wish to fragment the output packet. In this case, the packet handler code will attempt to fragment any response greater than 16 octets in size.

```
/* now fragment the output */
/* set up the frag header */
fhdr = (struct ip6_frag *) &out_packet_buffer[40];

/* the next header is a UDP packet header */
fhdr->ip6f_nxt = 17 ;
fhdr->ip6f_reserved = 0 ;
fhdr->ip6f_offlg = htons(1);
fhdr->ip6f_ident = rand() % 4294967296 ;

/* the total size to send is the payload size plus the UDP header */
to_send = dns_response->len + 8 ;
to_buf = (char *) uhdr ;

/* now carve up the UDP response into frags */
/* initial block of UDP payload size is at least 8 bytes less than the original dns response */
units = dns_response->len / 8 ;
if (units > 16) datalen = 128 ;
else datalen = (units - 1) * 8 ;
frag_offset = 0 ;

/* add in the size of the UDP header into datalen in the first instance */
datalen += 8 ;

/* Destination and Source MAC addresses */
memcpy(ether_frame, dst_mac, 6 * sizeof (uint8_t));
memcpy(ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

/* Next is ethernet type code (ETH_P_IPV6 for IPv6) */
ether_frame[12] = ETH_P_IPV6 / 256;
ether_frame[13] = ETH_P_IPV6 % 256;

/* now send the payload in fragments */
while (to_send > 0) {

  /* each time we send datalen bytes plus the 8 byte frag header */
  iphdr->ip6_plen = htons(datalen + 8);

  /* now assemble the ether frame  using 2 x 6 octet MAC addresses, a 2 octet
     Ethernet frame type field, a 40 octet IPv6 header, a 8 octet Extension
     Header and the payload */
  frame_length = 6 + 6 + 2 + 40 + 8 + datalen;

  /* IPv6 header + frag header */
  memcpy (ether_frame + ETH_HDRLEN, iphdr, 48);

  /* payload fragment */
  memcpy (ether_frame + ETH_HDRLEN + 48, to_buf, datalen);

  /* Send ethernet frame out using the raw socket */
  if ((bytes = sendto (sd, ether_frame, frame_length, 0,
          (struct sockaddr *) &device, sizeof (device))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);
    }
  to_send -= datalen ;
  to_buf += datalen ;

  if (to_send > 0) {
    if (to_send <= 512) {

      /* last frag */
      frag_offset += (datalen / 8) ;
      fhdr->ip6f_offlg = htons(frag_offset << 3) ;
      datalen = to_send ;
      }
    else {
      frag_offset += (datalen / 8) ;
      fhdr->ip6f_offlg = htons((frag_offset << 3) + 1);
      datalen = 512 ;
      }
    }
  }
```

It should be noted that this is only a UDP interceptor, and if a client attempts to connect using TCP, then this packet handler will not be invoked A complete DNS front end would also perform TCP interception. However, this is not so straight forward as UDP, as we will see when we look at a fragmenting TCP packet handler in the next section.

The complete code for this UDP DNS packet handler can be found in a GitHub repository (https://github.com/gih900/IPv6--DNS-Frag-Test-Rig).

A Raw Socket TCP HTTP(S) Server

The second part of our experiment was to measure the drop rate of fragmented IPv6 packets that were directed towards end users. Within the constraints of our experimental setup this implied that we need to place a packet fragmentation module into a TCP session. The approach taken here is similar to that used for UDP, namely using a 'normal' HTTP(S) server as a backend server, and implementing a packet handler to perform the fragmentation of outbound packets directed to the end client.

There are a number of possible implementation options. The chosen option was to use a stateless packet handler along the lines of an IPv6 NAT. Incoming SYN packets from the client create a new NAT binding state and the packet headers are re-written so that the packet destination address is that of the back end HTTP processor and the packet source address is this host. Other incoming packets from the client were matched against an established NAT binding state, the packet headers were re-written and passed to the back end. Packets from the back end needed to be matched against an established NAT state, and the packet headers were translated and large payload responses were fragmented as they were passed to the client. This design is shown in Figure 2.

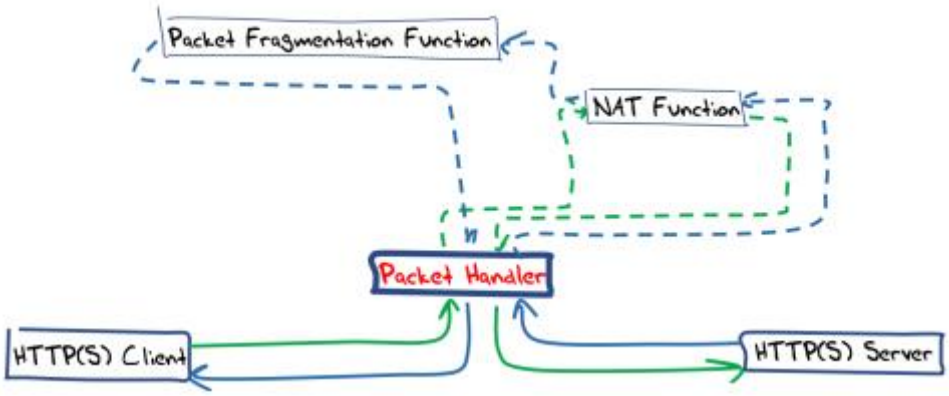

Figure 2 – TCP Packet Handler

The first task is to receive TCP packets from the client and get them passed to the application, bypassing the operating system's conventional behaviour. Most Unix systems will recognise an incoming TCP packet and if there is no listening TCP port associated with the destination port number, the system will respond with a TCP reset (RST) packet. If there is a TCP session associated with the port, then the operating system's IP and TCP drivers will process the packet and our raw packet handlers will not be invoked in any case. We need to stop this automatic RST generation when receiving unbound TCP packets.

In FreeBSD systems there is a system parameter than can be set to turn off this behaviour:

```
sysctl -w net.tcp.blackhole=2.
```

Debian Linux systems don't appear to have this form of control over these kernel-generated TCP reset responses, so the alternative is to suppress the TCP reset packet before it leaves the system. The approach we used was to set up a filter in outbound packets using an `iptables` filter entry:

```
ip6tables -I OUTPUT 1 -o eth0 -p tcp --tcp-flags ALL RST -j DROP
```

Strictly speaking this does not suppress the generation of the RST packet, but stops it leaving the system, which is the same overall result as suppression in any case.

We then need to pick up all incoming IPv6 TCP packets addressed to the listening address on port 80 and port 443 and process them within the context of this application. The simplest way to achieve this is by using the packet capture library routines (`libpcap`).

```
/* the PCAP capture filter  - http and https v6 traffic only*/
sprintf(filter_exp,"dst host %s and (port 80 or port 443) and tcp and ip6",host);

/* open capture device */
if ((handle = pcap_open_live(interface, SNAP_LEN, 1, 1, errbuff)) == NULL)   {
  fprintf(stderr, "Couldn't open device %s: %s\n", interface, errbuff);
  exit(EXIT_FAILURE) ;
  }

/* compile the filter expression */
if (pcap_compile(handle, &fp, filter_exp, 0, 0) == -1) {
  fprintf(stderr, "Couldn't parse filter %s: %s\n", filter_exp, pcap_geterr(handle));
  exit(EXIT_FAILURE) ;
  }

/* install the filter */
if (pcap_setfilter(handle, &fp) == -1) {
  fprintf(stderr, "Couldn't install filter %s: %s\n", filter_exp, pcap_geterr(handle));
  exit(EXIT_FAILURE) ;
  }

/* set up the packet capture in an infinite loop */
if (debug) printf("Enter PCAP packet capture loop\n") ;
pcap_loop(handle, -1, got_packet, NULL) ;
```

Each incoming packet will invoke the packet handler `got_packet()`, which can be used to process both packets coming in from the client and packets coming in from the HTTP(S) back end.

However, before looking at this code there is one more aspect of many modern host systems that we need to disable. These days many Ethernet interfaces are "smart" and part of that additional functionality is to perform TCP segmentation offloading into the interface card. The argument in favour of this is one of relieving the kernel of extraneous I/O interrupts, allowing the kernel to pass large buffers of memory to the interface card and have the card perform TCP segmentation for outgoing packets. Similarly, the card may aggregate several arriving packets into a single pseudo-TCP segment that is passed to the operating system in a single interrupt transaction. Obviously, we need to disable this functionality in this context. The `ethtool` package provides a way to manage these I/O devices:

```
ethtool -K eth0 generic-segmentation-offload off
ethtool -K eth0 generic-receive-offload off
```

We can now examine the major aspects of the packet handler within the `got_packet()` routine. The first is the handling of packets that come from the end client. The source address and source port are used to lookup the local NAT translation table, and this is used to create an outbound packet destined to the back-end HTTP(S) server.

```
/* this is a packet from the client to the packet handler */
if ((dport == 80) || (dport === 443)) {

  /* search of a matching entry in the local NAT table using the
     source address and source port as the lookup key */
  bcopy(&(ip->ip6_src), &(bdg.ip6_src),16);
  bdg.sport = sport ;
  bdp = find_binding(&bdg,dport) ;
```

```
  /* if we cannot find a NAT table entry, and the packet contains a TCP
     SYN flag, then we can create a new NAT entry */
  if ((!bdp) && (tcp->th_flags & TH_SYN)) {
    if (dport == 80) {
      port = next_port_80(&(ip->ip6_src),ntohs(tcp->th_sport)) ;
      bdp = port_80_ptr[port]->entry;
      }
    else {
      port = next_port_443(&(ip->ip6_src),ntohs(tcp->th_sport)) ;
      bdp = port_443_ptr[port]->entry;
      }
    }
  else if (!bdp)
    return;

  /* retain the current TCP session sequence number in the NAT table */
  bdp->seq = tcp->th_seq ;

  /* perform a header translation and pass the packet to the back-end
     http(s) server */
  send_packet_to_http(packet,bdp)  ;
  }
```

The packets received from the back end have the 'reverse' NAT header substitution applied in both the IPv6 and TCP packet headers. The TCP checksum on the TCP pseudo-header is computed and the response is ready for the final phase of sending.

```
/* IPv6 header */
iphdr = (struct ip6_hdr *) &out_packet_buffer[0] ;
o_iphdr = (struct ip6_hdr *) (packet + ETH_HDRLEN) ;

/* IPv6 version (4 bits), Traffic class (8 bits), Flow label (20 bits) */
iphdr->ip6_flow = o_iphdr->ip6_flow ;


/* payload length */
len = ntohs(o_iphdr->ip6_plen);

/* Hop limit (8 bits): default to maximum value */
iphdr->ip6_hops = 255;

/* src address */
bcopy(&local6_addr,&(iphdr->ip6_src), 16) ;

/* dst address */
bcopy(&tp->ip6_src,&(iphdr->ip6_dst), 16);

/* TCP header */
orig_tcp = (struct tcphdr *) (packet + ETH_HDRLEN + 40);
tcp = (struct tcphdr *) &(out_packet_buffer[40]);
tcp->th_dport =  htons(tp->sport) ;
tcp->th_sport = orig_tcp->th_sport;

/* copy payload bytes from the original packet to the payload buffer */
memcpy(&out_packet_buffer[44],&packet[ETH_HDRLEN + 44],len - 4) ;

/* Destination and Source MAC addresses */
memcpy(ether_frame, dst_mac, 6 * sizeof (uint8_t));
memcpy(ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

/* Next is ethernet type code (ETH_P_IPV6 for IPv6) */
ether_frame[12] = ETH_P_IPV6 / 256;
ether_frame[13] = ETH_P_IPV6 % 256;


/* Copy the IPv6 header into the ether_frame */
memcpy(ether_frame + ETH_HDRLEN, &out_packet_buffer[0], 40);

payload = ntohs(iphdr->ip6_plen) - (tcp->th_off * 4) ;
pptr = &out_packet_buffer[40 + (tcp->th_off * 4)];
tcp_hdr_len = tcp->th_off * 4 ;
tcp_sequence = ntohl(tcp->th_seq) ;
```

If the payload is small we'll send it as is, without additional fragmentation.

```
if (payload <= 16) {
  /* copy across the TCP header */
```

```
memcpy(ether_frame + ETH_HDRLEN + 40, &out_packet_buffer[40], tcp_hdr_len);

/* copy across the payload */
memcpy(ether_frame + ETH_HDRLEN + 40 + tcp_hdr_len, pptr, payload);

/* IPv6 payload length */
e_iphdr = (struct ip6_hdr *) (ether_frame + ETH_HDRLEN) ;
e_iphdr->ip6_plen = htons(tcp_hdr_len + payload);

/* Next header (8 bits): 6 for TCP */
e_iphdr->ip6_nxt = IPPROTO_TCP;

/* put in the adjusted sequence number and the new checksum */
e_tcp = (struct tcphdr *) (ether_frame + ETH_HDRLEN + 40) ;
e_tcp->th_seq = htonl(tcp_sequence) ;
e_tcp->th_sum = 0 ;
e_tcp->th_sum = tcp_checksum(e_tcp,tcp_hdr_len+payload,tcp_hdr_len+payload,
                                    &(e_iphdr->ip6_src),&(e_iphdr->ip6_dst)) ;
/* ethernet frame length */
frame_length = ETH_HDRLEN + 40 + tcp_hdr_len + payload ;

if ((bytes = sendto (sd, ether_frame, frame_length, 0,
           (struct sockaddr *) &device, sizeof (device))) <= 0) {
  perror ("sendto() failed");
  exit (EXIT_FAILURE);
  }
return;
}
```

We are using the raw socket `sd` to send this packet. Given that we are already using the `pcap` library to receive these packets directly off the interface an alternative would be to use the `pcap_sendpacket()` routine instead. This alternative approach would be more portable across operating system platforms, as it would no longer be reliant on a particular form of interface control functions that are only exposed in Linux systems, but perhaps this alternative approach is best left as an exercise for an enthusiastic reader.

For larger payloads we'll perform re-segmentation and fragmentation. We'll re-segment each packet so that each TCP packet it is no larger than 1,248 octets, and then we'll fragment the packet into two parts, with the trailing frag containing the last 256 octets of the payload.

```
while (payload > 0) {

  /* lets pull off the trailing 8 (or so bytes) into a fragmented trailer */
  if (payload < 1200) {
    tcp_seg_len = payload ;
    this_frag = ((payload / 8) - 1) * 8 ;
    }

  /* lets re-segment the packet */
  if (payload >= 1200) {
    tcp_seg_len = 1200 ;
    this_frag = 1200 - 256 ;
    }

  /* start constructing the IPv6 header in the Ethernet frame
     by writing in the IPv6 payload length and next header fields */
  e_iphdr = (struct ip6_hdr *) (ether_frame + ETH_HDRLEN) ;
  e_iphdr->ip6_plen = htons(8 + tcp_hdr_len + this_frag);
  e_iphdr->ip6_nxt = 44 ; // Fragmentation header

  /* now set up the frag header */
  fhdr = (struct ip6_frag *) &ether_frame[ETH_HDRLEN+40] ;
  fhdr->ip6f_nxt = IPPROTO_TCP ;
  fhdr->ip6f_reserved = 0 ;
  fhdr->ip6f_offlg = htons(1); // Offset is zero and Set more-fragments flag
  fhdr->ip6f_ident = rand() % 4294967296 ;

  /* for the first frag, copy across the TCP header */
  memcpy(ether_frame + ETH_HDRLEN + 40 + 8, &out_packet_buffer[40], tcp_hdr_len);

  /* copy across the entire payload (in order to generate the correct tcp checksum) */
  memcpy(ether_frame + ETH_HDRLEN + 40 + 8 + tcp_hdr_len, pptr, tcp_seg_len);

  /* put in the adjusted sequence number and the new checksum */
  e_tcp = (struct tcphdr *) (ether_frame + ETH_HDRLEN + 40 + 8) ;
  e_tcp->th_seq = htonl(tcp_sequence) ;
```

```
e_tcp->th_sum = 0 ;
e_tcp->th_sum = tcp_checksum(e_tcp,tcp_hdr_len + tcp_seg_len, tcp_hdr_len + tcp_seg_len,
                      &(e_iphdr->ip6_src),&(e_iphdr->ip6_dst)) ;

/* send the leading fragment to the Ethernet interface */
frame_length = ETH_HDRLEN + 40 + 8 + tcp_hdr_len + this_frag ;

if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *) &device,
             sizeof (device))) <= 0) {
  perror ("sendto() failed");
  exit (EXIT_FAILURE);
  }

pptr += this_frag;
offset = (this_frag + tcp_hdr_len) >> 3;
remainder = tcp_seg_len - this_frag ;

/* now adjust the frag header for the trailing frag */
fhdr->ip6f_offlg = htons(offset << 3); // Offset
this_frag = remainder ;
e_iphdr->ip6_plen = htons(8 + this_frag);

/* copy across the remainder of the payload immediately folloing the frag header */
memcpy(ether_frame + ETH_HDRLEN + 40 + 8, pptr, this_frag);


/* send the trailing fragment */
frame_length = ETH_HDRLEN + 40 + 8 + this_frag ;

if ((bytes = sendto (sd, ether_frame, frame_length, 0, (struct sockaddr *) &device,
             sizeof (device))) <= 0) {
  perror ("sendto() failed");
  exit (EXIT_FAILURE);
  }

pptr += this_frag ;
tcp_sequence += tcp_seg_len ;
payload -= tcp_seg_len ;
}
```

The complete code for this TCP packet handler can be found in a GitHub repository (https://github.com/gih900/IPv6--TCP-Frag-Test-Rig).

I find it to be a little odd that it's so challenging to set up a raw socket interface in IPv6. It seems that the challenges start with interface cards that by default want to perform TCP segmentation and UDP fragmentation on output and perform the reverse function on input. Then we have some operating systems that seem to be hardwired to send reset packets when receiving TCP packets that to not address an open TCP socket. Then we have the IPv6 protocol engine that simply does not support raw socket connections in the same manner as IPv4.

The good news, however, is that an equivalent level of functionality can be coerced to work on most systems, and even where it does not appear that the ethernet level socket calls are supported, the *libpcap* library can be used in its place, as this library supports both sending and receiving packets.

Author

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region.

www.potaroo.net

Disclaimer

The above views do not necessarily represent the views or positions of the Asia Pacific Network Information Centre.