# EtherLeak:
# Ethernet frame padding information leakage

By Ofir Arkin and Josh Anderson

**Abstract**

Multiple platform Ethernet Network Interface Card (NIC) device drivers incorrectly handle frame padding, allowing an attacker to view slices of previously transmitted packets or portions of kernel memory. This vulnerability is the result of incorrect implementations of RFC requirements and poor programming practices, the combination of which results in several variations of this information leakage vulnerability. This bug is explored in its various manifestations through code examples and packet captures. Solutions to this flaw are provided.

## Contents

## 1.0 Introduction

A link layer implementation flaw in multiple platforms allows an attacker to view slices of previously transmitted packets or portions of kernel memory. This vulnerability arises when bytes used to pad an Ethernet frame to the minimum size are copied directly from a dirty buffer without being sanitized. This information leakage vulnerability is trivial to exploit and has potentially devastating consequences. Several different variants of this implementation flaw result in this vulnerability; all of them will be examined in depth within this paper. The Linux, NetBSD and Microsoft Windows operating systems are known to have vulnerable link layer implementations and it is extremely likely that other operating systems are also affected.

The Ethernet standards impose strict limitations on the size of encapsulated packets, requiring small packets to be padded up to a minimum size. Many device drivers responsible for Ethernet frame generation incorrectly handle the padding of small packets. They use data from the end of the frame manipulation buffer without initializing it to zero as required by the RFCs. The location of the buffer containing the frame determines the contents of the padding bytes.

The next sections provide a brief introduction to Ethernet and Ethernet frames. Following this is a detailed examination of the flaw utilizing examples, focusing on the various implementations of this bug. Finally, solutions and areas of suggested further investigation will be presented.

## 2.0 A Brief Introduction to Ethernet

This section provides a brief introduction to Ethernet, outlining the protocol's history, the data transmission structures and how they interface with higher layer protocols.

### 2.1 History

Ethernet was first developed at Xerox labs in Palo Alto, California, in 1973. Seven years later the DEC-Intel-Xerox (DIX) consortium, formed to standardize and commercialize Ethernet technology, published the first 10Mbps Ethernet standard. This standard is commonly referred so as "Ethernet version I", and a revision of the standard, published in 1982, is referred to as "Ethernet version II". This last version of the standard is frequently referred to as the "DIX Ethernet Standard". In 1985 the IEEE published "Carrier Sense Multiple Access with Collision Detection Access Method and Physical Layer Specifications", better known as the 802.3 CSMA/CD standard. Based on the DIX Ethernet standard the IEEE 802.3 standard is currently the official Ethernet standard. Although adding a number of technical changes, the 802.3 standard remains backward compatible with the original DIX Ethernet standard. Since publishing 802.3 the IEEE has standardized a number of Ethernet related technologies such as 100Mbps Fast Ethernet, Gigabit Ethernet and others.

### 2.2 Ethernet Frames

Ethernet is a link layer protocol used to transmit higher layer protocols between hosts. Data transition across the Ethernet is formed of discrete streams of bits, called frames. These frames must conform to one of two formats defined within the 802.3 standard. Frame formats enable a receiving system to correctly interpret a bit stream from the Ethernet as intelligible data organized into sections, called fields. Despite technical differences between the two formats they share several common fields. These fields provide some protection against data loss, detail the origin and destination of the frame, contain the encapsulated data and provide other important information.

The data field is a variable length buffer that contains the encapsulated higher-layer packet being transmitted by the frame over the Ethernet. Restrictions within the Ethernet standards limit the size of this field to a minimum of 46 bytes and a maximum of 1500 bytes. If the higher level protocol data is less than the minimum it must be padded to the correct size.

The two frame formats differ primarily in the mechanism they use to describe their encapsulated data, which gives rise to their names "Type encapsulation" and "Length encapsulation". The Type encapsulation format describes the contents of its data field by the type of the higher layer protocol packet it encapsulates. The Length encapsulation format describes the length of the data field, and uses an optional LLC header to describe the higher layer protocol type.

## 2.3 Interfacing TCP/IP with Ethernet Frames

The encapsulation of IP packets for transmission over Ethernet networks is defined in two RFCs. RFC 894[1], "A Standard for the Transmission of IP Datagrams over Ethernet Networks", defines the encapsulation of IP packets for the DIX Ethernet standard and RFC 1042[2], "A Standard for the Transmission of IP Datagrams over IEEE 802 Networks", defines the encapsulation of IP packets for IEEE 802 networks. Both RFCs provide instructions on how to handle IP packets smaller than the minimum data field size (46 bytes) required by the Ethernet standards. They require that "the data field should be padded (with octets of zero) to meet the Ethernet minimum frame size". This padding is not part of the IP packet and should not be included in the total length field of the IP header; it is simply part of the link-layer.

Ambiguities within the RFCs leave it unclear as to who is responsible for padding the Ethernet frames; as such it is implementation dependent. Various different systems have implemented this padding operation in places such as the network interface card hardware ("auto padding"), the software device driver and even in a separate layer 2 stack. Several implementation flaws have been discovered in the padding functionality of many device drivers. These flaws will be examined in the following section.

## 3.0 The Ethernet Frame Padding Bug

This section will examine the programming error behind the bug. The effects of each variation of this error will be analyzed through examples of offending code and frame captures originating from vulnerable drivers. Before delving into complex device driver code a generic example of the bug will be examined to clearly demonstrate the core programming mistake.

### 3.1 Theory

The following example presents the basic programming error which causes this information leakage vulnerability.

```
01 void xmit_frame(char *frame_buf, int frame_len)
02 {
03    int   length;
04
05
06    if (frame_len < MIN_FRAME_SZ)
07          length = MIN_FRAME_SZ;
08    else
09          length = frame_len;
10
11    copy_to_tx_buf(frame_buf, length);
12
13    return;
14 }
```

The function `xmit_frame()` takes two parameters, the first is a buffer containing the frame to be transmitted, and the second contains the length of the frame within the buffer. The code performs checks to ensure that the frame meets the minimum length requirements of the Ethernet standards (line 06). If the length is less than this minimum then `length` is initialized to the minimum (line 07), otherwise `length` is assigned the actual frame length from `frame_len` (line 09). The value of `length` is used to copy the frame buffer out to the transmit buffer on the network device (line 11). At this point an undersized transmitted frame will be padded, from `frame_len` to `length`, with the previous contents of the buffer. This vestigial data is the source of the information leakage.

The fix for this example code, and indeed the majority of the affected code, is simply an additional line to zero out any padding bytes.  An example solution is given below.

```
...
06    if (frame_len < MIN_FRAME_SZ) {
07          length = MIN_FRAME_SZ;
08          memset(frame_buf + frame_len, 0, length - frame_len);
09    } else
...
```

This example code makes the programming mistake obvious to even a non-programmer, but, surprisingly, many production device drivers contain similarly flawed code. The error is essentially the same for all of the code examined during the writing of this paper, but its effects differ depending on the location of the buffer containing the frame prior to transmission. There are basically three potential buffer locations from which the pad bytes can originate:

- Dynamic kernel buffer
- Static device driver buffer
- Hardware device transmit buffer

This section examines each of these.

Please note in the following code examples only the relevant lines have been numbered, and although the authors refer to `ETH_ZLEN` as the minimum Ethernet frame size its value is in fact implementation dependent. Different devices provide different services for outgoing frames and their corresponding drivers behave accordingly.

## 3.2 Dynamic Kernel Buffer

Ethernet frames are typically constructed around an encapsulated higher layer protocol packet. These packets are built by the kernel TCP/IP stack and are manipulated within buffers allocated for this purpose. This example is from the Linux 2.4.18 kernel which uses a single buffer per packet. The buffer is dynamically allocated from kernel memory when required and thus its contents, if not initialized, will contain vestigial data from the kernel memory pool.

### 3.2.1 Driver Code

`/usr/src/linux/drivers/net/atp.c`[1]:

```
static int atp_send_packet(struct sk_buff *skb, struct net_device *dev)
{
        struct net_local *lp = (struct net_local *)dev->priv;
        long ioaddr = dev->base_addr;
        int length;
        long flags;

01      length = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN;

        netif_stop_queue(dev);

       /* Disable interrupts by writing 0x00 to the Interrupt Mask Register.
          This sequence must not be interrupted by an incoming packet. */

        spin_lock_irqsave(&lp->lock, flags);
        write_reg(ioaddr, IMR, 0);
        write_reg_high(ioaddr, IMR, 0);
        spin_unlock_irqrestore(&lp->lock, flags);
```

[1] atp.c is a Linux device driver for Ethernet adapters based on the RealTek RTL8002 and RTL8012 chipsets.

```
02      write_packet(ioaddr, length, skb->data, dev->if_port);

        lp->pac_cnt_in_tx_buf++;
        if (lp->tx_unit_busy == 0) {
                trigger_send(ioaddr, length);
                lp->saved_tx_size = 0;               /* Redundant */
                lp->re_tx = 0;
                lp->tx_unit_busy = 1;
        } else
                lp->saved_tx_size = length;
        /* Re-enable the LPT interrupts. */
        write_reg(ioaddr, IMR, ISR_RxOK | ISR_TxErr | ISR_TxOK);
        write_reg_high(ioaddr, IMR, ISRh_RxErr);

        dev->trans_start = jiffies;
        dev_kfree_skb (skb);
        return 0;
}
```

The `skb` structure contains a pointer to the frame construction buffer at `skb->data` and the size of the frame within that buffer at `skb->len`. The code at line 01 uses a C language construct to initialize `length` to the greater of the two size values: `ETH_ZLEN` or `skb->len`. Therefore, if the length of the frame is less than the Ethernet minimum (`ETH_ZLEN`) it will be rounded up to that minimum. The value of `length` is then used when copying the frame out from the `skb->data` buffer to the hardware device (line 02).

The contents of the pad bytes is heavily dependent on system load. Due to the large address space of kernel memory and small number of pad bytes it is extremely unlikely that sensitive information will be leaked. However the origin of these pad bytes makes any such information potentially devastating.

### 3.2.2 Frame Capture

The following packet capture demonstrates how the contents of the pad bytes change as different portions of kernel memory are used for the construction of ICMP echo reply packets.

```
14:18:48.146095 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                            4500 001d 0000 4000 4001 240c 0a32 013c
                            0a32 0135 0800 83f9 5206 2200 00
14:18:48.146393 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id 3747, len 29)
                            4500 001d 0ea3 0000 ff01 9668 0a32 0135
                            0a32 013c 0000 8bf9 5206 2200 0059 0100
                            6c02 0000 c006 0000 0600 0000 0010
14:18:49.146095 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                            4500 001d 0000 4000 4001 240c 0a32 013c
                            0a32 0135 0800 82f9 5206 2300 00
14:18:49.146387 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id 3749, len 29)
                            4500 001d 0ea5 0000 ff01 9666 0a32 0135
                            0a32 013c 0000 8af9 5206 2300 0000 4003
                            2300 4003 f101 1200 ed01 feff 0000
```

```
14:18:50.146093 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                           4500 001d 0000 4000 4001 240c 0a32 013c
                           0a32 0135 0800 81f9 5206 2400 00
14:18:50.146396 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id
3751, len 29)
                           4500 001d 0ea7 0000 ff01 9664 0a32 0135
                           0a32 013c 0000 89f9 5206 2400 0000 8002
                           1700 e002 1700 4003 9101 c001 0600
```

## 3.3 Static Device Driver Buffer

Some device drivers copy frames into internal buffers prior to sending them to the network card. The reasons for doing this vary from card to card. This example code for Linux 2.4 uses an alignment buffer to ensure that the frame buffer's address is digestible by the card's hardware. The RealTek 8139 chipset requires that the address of the buffer be aligned on a 32-bit `word` boundary. Because this reusable alignment buffer is not cleaned between uses, the contents of the previous occupant will be inadvertently used as pad bytes.

### 3.3.1 Driver Code

ftp://www.scyld.com/pub/network/test/rtl8139.c:[2]

```
static int
rtl8129_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct rtl8129_private *tp = (struct rtl8129_private *)dev->priv;
    long ioaddr = dev->base_addr;
    int entry;

    if (netif_pause_tx_queue(dev) != 0) {
            /* This watchdog code is redundant with the media monitor timer.
*/
            if (jiffies - dev->trans_start > TX_TIMEOUT)
                    rtl8129_tx_timeout(dev);
            return 1;
    }

    /* Calculate the next Tx descriptor entry. */
    entry = tp->cur_tx % NUM_TX_DESC;

    tp->tx_skbuff[entry] = skb;
01  if ((long)skb->data & 3) {          /* Must use alignment buffer. */
02          memcpy(tp->tx_buf[entry], skb->data, skb->len);
03          outl(virt_to_bus(tp->tx_buf[entry]), ioaddr + TxAddr0 +
entry*4);
04  } else
05          outl(virt_to_bus(skb->data), ioaddr + TxAddr0 + entry*4);
    /* Note: the chip doesn't have auto-pad! */
06  outl(tp->tx_flag | (skb->len >= ETH_ZLEN ? skb->len : ETH_ZLEN),
             ioaddr + TxStatus0 + entry*4);

    /* There is a race condition here -- we might read dirty_tx, take an
```

[2] rtl8139.c is a Linux device driver for the RealTek 8129/8139 chipsets.

```
       interrupt that clears the Tx queue, and only then set tx_full.
       So we do this in two phases. */
    if (++tp->cur_tx - tp->dirty_tx >= NUM_TX_DESC) {
             set_bit(0, &tp->tx_full);
             if (tp->cur_tx - (volatile unsigned int)tp->dirty_tx <
NUM_TX_DESC) {
                      clear_bit(0, &tp->tx_full);
                      netif_unpause_tx_queue(dev);
             } else
                      netif_stop_tx_queue(dev);
    } else
             netif_unpause_tx_queue(dev);

    dev->trans_start = jiffies;
    if (tp->msg_level & NETIF_MSG_TX_QUEUED)
             printk(KERN_DEBUG"%s: Queued Tx packet at %p size %d to slot
%d.\n",
                         dev->name, skb->data, (int)skb->len, entry);

    return 0;
}
```

The code here is similar to the previous example differing only in the check to ensure that the buffer containing the frame is properly aligned. If the address of the buffer is not aligned on a four byte boundary (line 01) then the frame will be copied to a buffer maintained by the device driver which is aligned (line 02). The address of the aligned buffer, either the driver's alignment buffer (line 03) or the original buffer (line 05), is passed to the card. The card is informed of the address of the frame buffer and the length of the frame at line 06. The greater of the two values, `ETH_ZLEN` or `skb->len`, is passed to the card as the size of the frame to transmit (line 06).

The pad bytes will always contain the contents of earlier packets and therefore present a great likelihood of revealing sensitive data.

### 3.3.2 Frame Capture

The following example was produced by a Linux 2.4.x machine with a Xircom PCMCIA network card using the `xirc2ps_cs.c` device driver.

The following `tcpdump` traces clearly demonstrates the information leakage problem. The information leaked is HTTP-based traffic.

```
14:49:09.306008 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 577, len 28)
0x0000  4500 001c 0241 4000 8001 bb29 c0a8 de0b       E....A@....)....
0x0010  c0a8 de19 0000 48f9 b406 0300 0400 0000       ......H.........
0x0020  0004 0650 5542 4c49 4303 4e4c 5307            ...PUBLIC.NLS.
14:50:46.313706 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 854, len 28)
0x0000  4500 001c 0356 4000 8001 ba14 c0a8 de0b       E....V@.........
0x0010  c0a8 de19 0000 e7f8 b406 6400 4b43 4c41       ..........d.KCLA
0x0020  4e47 2e44 4c4c 3c00 0090 2787 ce24            NG.DLL<...'..$
14:51:02.315077 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 870, len 28)
0x0000  4500 001c 0366 4000 8001 ba04 c0a8 de0b       E....f@.........
0x0010  c0a8 de19 0000 d7f8 b406 7400 7468 656d       ..........t.them
0x0020  652e 646c 6c3c 0000 9027 87ce 2400            e.dll<...'..$.
```

```
14:51:11.315842 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 895, len 28)
0x0000  4500 001c 037f 4000 8001 b9eb c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 cef8 b406 7d00 743a 204d       ..........}.t:.M
0x0020  6f7a 696c 6c61 2f35 2e30 2028 5769            ozilla/5.0.(Wi
14:51:50.318904 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1435, len 28)
0x0000  4500 001c 059b 4000 8001 b7cf c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 a7f8 b406 a400 6b65 6570       ............keep
0x0020  2d61 6c69 7665 0d0a 436f 6f6b 6965            -alive..Cookie
14:51:51.319076 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1436, len 28)
0x0000  4500 001c 059c 4000 8001 b7ce c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 a6f8 b406 a500 434f 500d       ............COP.
0x0020  0a52 6566 6572 6572 3a20 6874 7470            .Referer:.http
14:52:03.320000 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1449, len 28)
0x0000  4500 001c 05a9 4000 8001 b7c1 c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 9af8 b406 b100 2057 696e       .............Win
0x0020  646f 7773 204e 5420 352e 303b 2065            dows.NT.5.0;.e
14:52:04.320125 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1450, len 28)
0x0000  4500 001c 05aa 4000 8001 b7c0 c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 99f8 b406 b200 2f78 6d6c       ............/xml
0x0020  2c61 7070 6c69 6361 7469 6f6e 2f78            ,application/x
14:52:05.320151 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1451, len 28)
0x0000  4500 001c 05ab 4000 8001 b7bf c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 98f8 b406 b300 742f 706c       ............t/pl
0x0020  6169 6e3b 713d 302e 382c 7669 6465            ain;q=0.8,vide
14:52:06.320274 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1452, len 28)
0x0000  4500 001c 05ac 4000 8001 b7be c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 97f8 b406 b400 2c74 6578       ............,tex
0x0020  742f 6373 732c 2a2f 2a3b 713d 302e            t/css,*/*;q=0.
14:52:07.320319 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1453, len 28)
0x0000  4500 001c 05ad 4000 8001 b7bd c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 96f8 b406 b500 677a 6970       ............gzip
0x0020  2c20 6465 666c 6174 652c 2063 6f6d            ,.deflate,.com
14:52:08.320393 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1455, len 28)
0x0000  4500 001c 05af 4000 8001 b7bb c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 95f8 b406 b600 436f 6f6b       ............Cook
0x0020  6965 3a20 4153 5053 4553 5349 4f4e            ie:.ASPSESSION
14:52:09.320478 192.168.222.11 > 192.168.222.25: icmp: echo reply (DF) (ttl
128, id 1456, len 28)
0x0000  4500 001c 05b0 4000 8001 b7ba c0a8 de0b       E.....@.........
0x0010  c0a8 de19 0000 94f8 b406 b700 3a20 6874       ............:.ht
0x0020  7470 3a2f 2f77 7777 2e63 7279 7374            tp://www.cryst
```

## 3.4 Hardware Device Transmit Buffer

Some device drivers contain a particularly severe variation of this bug in which they incorrectly inform the card of the number of bytes which have been placed into the hardware transmit buffer. When this happens, the padding bytes are supplied by the contents of the previous frame occupying that space. The Linux 2.4 `axnet_cs.c` device driver demonstrates just such a variant. The code incorrectly informs the device that the minimum number of bytes have been copied out to the Tx buffer, when in fact the actual length of the frame is all that has been supplied. The card will transmit the length it has been informed of, and thus send bytes from its Tx buffer as frame padding.

### 3.4.1 Driver Code

`/usr/src/linux/drivers/net/pcmcia/axnet_cs.c`[3]:

```
static int ei_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
        long e8390_base = dev->base_addr;
        struct ei_device *ei_local = (struct ei_device *) dev->priv;
        int length, send_length, output_page;
        unsigned long flags;

        netif_stop_queue(dev);

01      length = skb->len;

        /* Mask interrupts from the ethercard.
           SMP: We have to grab the lock here otherwise the IRQ handler
           on another CPU can flip window and race the IRQ mask set. We end
           up trashing the mcast filter not disabling irqs if we dont lock
*/

        spin_lock_irqsave(&ei_local->page_lock, flags);
        outb_p(0x00, e8390_base + EN0_IMR);
        spin_unlock_irqrestore(&ei_local->page_lock, flags);

        /*
         *      Slow phase with lock held.
         */
        disable_irq_nosync(dev->irq);

        spin_lock(&ei_local->page_lock);

        ei_local->irqlock = 1;

02      send_length = ETH_ZLEN < length ? length : ETH_ZLEN;

        /*
         * We have two Tx slots available for use. Find the first free
         * slot, and then perform some sanity checks. With two Tx bufs,
         * you get very close to transmitting back-to-back packets. With
         * only one Tx buf, the transmitter sits idle while you reload the
         * card, leaving a substantial gap between each transmitted packet.
         */

        if (ei_local->tx1 == 0)
        {
                output_page = ei_local->tx_start_page;
03              ei_local->tx1 = send_length;
                if (ei_debug  &&  ei_local->tx2 > 0)
                    printk(KERN_DEBUG "%s: idle transmitter tx2=%d,
lasttx=%d, txing=%d.\n",
                    dev->name, ei_local->tx2, ei_local->lasttx, ei_local-
>txing);
        }
        else if (ei_local->tx2 == 0)
        {
                output_page = ei_local->tx_start_page + TX_1X_PAGES;
04              ei_local->tx2 = send_length;
                if (ei_debug  &&  ei_local->tx1 > 0)
                        printk(KERN_DEBUG "%s: idle transmitter, tx1=%d,
lasttx=%d, txing=%d.\n",
                                dev->name, ei_local->tx1, ei_local->lasttx,
ei_local->txing);
        }
```

[3] axnet_cs.c is a Linux device driver for PCMCIA Ethernet cards based on the Asix88190 chipset.

```
        else
        {       /* We should never get here. */
                if (ei_debug)
                        printk(KERN_DEBUG "%s: No Tx buffers free! tx1=%d
tx2=%d last=%d\n",
                                dev->name, ei_local->tx1, ei_local->tx2,
ei_local->lasttx);
                ei_local->irqlock = 0;
                netif_stop_queue(dev);
                outb_p(ENISR_ALL, e8390_base + EN0_IMR);
                spin_unlock(&ei_local->page_lock);
                enable_irq(dev->irq);
                ei_local->stat.tx_errors++;
                return 1;
        }

        /*
         * Okay, now upload the packet and trigger a send if the transmitter
         * isn't already sending. If it is busy, the interrupt handler will
         * trigger the send later, upon receiving a Tx done interrupt.
         */

05      ei_block_output(dev, length, skb->data, output_page);
        if (! ei_local->txing)
        {
                ei_local->txing = 1;
                NS8390_trigger_send(dev, send_length, output_page);
                dev->trans_start = jiffies;
                if (output_page == ei_local->tx_start_page)
                {
                        ei_local->tx1 = -1;
                        ei_local->lasttx = -1;
                }
                else
                {
                        ei_local->tx2 = -1;
                        ei_local->lasttx = -2;
                }
        }
        else ei_local->txqueue++;

        if (ei_local->tx1  &&  ei_local->tx2)
                netif_stop_queue(dev);
        else
                netif_start_queue(dev);

        /* Turn 8390 interrupts back on. */
        ei_local->irqlock = 0;
        outb_p(ENISR_ALL, e8390_base + EN0_IMR);

        spin_unlock(&ei_local->page_lock);
        enable_irq(dev->irq);

        dev_kfree_skb (skb);
        ei_local->stat.tx_bytes += send_length;

        return 0;
}
```

The variable `length` is assigned the real length of the packet (line 01). The value of `length` is compared against the minimum frame length, and which ever is larger assigned to `send_length` (line 02). Depending on which of the two Tx buffers is free, the device is informed of the frame length at either line 03 or 04. However, at line 05 the actual length of the packet is used to copy the frame out to the device.

The padding bytes will always contain data from previous frames transmitted by the device. This is potentially the most severe manifestation of this bug because the data contained in the padding bytes is always from one of the last packets transmitted.

### 3.4.2 Frame Capture

The following example was produced by a Linux 2.4.x machine with a PCMCIA network card using the `axnet_cs.c` device driver.

The following `tcpdump` traces clearly demonstrates the information leakage problem.

```
14:18:48.146095 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                           4500 001d 0000 4000 4001 240c 0a32 013c
                           0a32 0135 0800 83f9 5206 2200 00
14:18:48.146393 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id 3747, len 29)
                           4500 001d 0ea3 0000 ff01 9668 0a32 0135
                           0a32 013c 0000 8bf9 5206 2200 0059 0100
                           6c02 0000 c006 0000 0600 0000 0010
14:18:49.146095 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                           4500 001d 0000 4000 4001 240c 0a32 013c
                           0a32 0135 0800 82f9 5206 2300 00
14:18:49.146387 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id 3749, len 29)
                           4500 001d 0ea5 0000 ff01 9666 0a32 0135
                           0a32 013c 0000 8af9 5206 2300 0000 4003
                           2300 4003 f101 1200 ed01 feff 0000
14:18:50.146093 10.50.1.60 > 10.50.1.53: icmp: echo request (DF) (ttl
64, id 0, len 29)
                           4500 001d 0000 4000 4001 240c 0a32 013c
                           0a32 0135 0800 81f9 5206 2400 00
14:18:50.146396 10.50.1.53 > 10.50.1.60: icmp: echo reply (ttl 255,
id 3751, len 29)
                           4500 001d 0ea7 0000 ff01 9664 0a32 0135
                           0a32 013c 0000 89f9 5206 2400 0000 8002
                           1700 e002 1700 4003 9101 c001 0600
```

## 4.0 Conclusion

A flaw in the handling of Ethernet frame padding has been presented and examined in detail. This link layer implementation flaw is known to affect multiple platforms and is believed to affect many more. The nature of this vulnerability makes it beyond the means of the authors to test and verify this bug on more than a limited number of systems, and the release of this paper is intended to prompt vendors to test and patch their products. The number of affected systems is staggering, and the number of vulnerable systems used as critical network infrastructure terrifying. The security of proprietary network devices is particularly questionable. Just how secure is your VLAN?

## 5.0 Acknowledgment

On May 28, 2002 Ofir Arkin released a bug report[3] concerning a layer 2 representation bug with `tcpdump`. Shortly after the release of this advisory, the authors were contacted by Stealth and Skyper of Team TESO and George Bakos (gbakos@ists.dartmouth.edu) of Dartmouth College, who provided them with examples of the `tcpdump` bug demonstrating information leakage. An investigation instigated by these reported examples led to the discovery of the bug. Throughout the writing of this paper George Bakos has been instrumental in testing and providing examples of this vulnerability. As such, the authors gratefully acknowledge the contributions of the above individuals. Additionally, the authors would like to thank the following people for contributions of networks, hardware and time:

- Bill Scherr
- Jeff Dike
- Vermont Army National Guard - Information Operations

## 6.0 References

[1] RFC 894 is available from: http://www.ietf.org/rfc/rfc894.txt
[2] RFC 1042 is available from: http://www.ietf.org/rfc/rfc1042.txt
[3] http://www.sys-security.com/archive/bugtraq/ofirarkin2002-03.txt

## Appendix A: Examples of Vulnerable Device Drivers

The following table lists some vulnerable device drivers from the Linux 2.4.18 Kernel. All device drivers can be found under `/usr/src/linux/drivers/net/`:

| Device Driver | Original Description |
|---|---|
| 3c501.c | A 3Com 3c501 Ethernet driver for Linux |
| 3c507.c | An EtherLink16 device driver for Linux |
| 3c523.c | net-3-driver for the 3c523 Etherlink/MC card (i82586 Ethernet chip) |
| 3c527.c | 3com Etherlink/MC32 driver for Linux 2.4 |
| 7990.c | LANCE Ethernet IC generic routines (AMD 7990 LANCE, local area network controller for Ethernet) |
| 8139too.c | RealTek RTL-8139 Fast Ethernet driver for Linux (based on rtl8139.c device driver which is also vulnerable) RTL 8129, 8139 chipsets |
| 82596.c | A generic 82596 Ethernet driver for Linux |
| 8390.c | A general NS8390 Ethernet driver core for Linux |
| a2065.c | Amiga Linux/68k A2065 Ethernet Driver |
| aironet4500_core.c | Aironet 4500/4800 driver core |
| am79c961a.c | driver for the am79c961A Lance chip used in the Intel (formally Digital Equipment Corp) EBSA110 platform. |
| ariadne.c | Amiga Linux/m68k Ariadne Ethernet Driver |
| arlan.c | This module provides support for the Arlan 655 card made by Aironet |
| at1700.c | A network device driver for the Allied Telesis AT1700 |
| atari_bionet.c | BioNet-100 device driver for linux68k |
| atarilance.c | Ethernet driver for VME Lance cards on the Atari |
| atari_pamsnet.c | PAMsNet device driver for linux68k |
| atp.c | Attached (pocket) Ethernet adapter driver for Linux (Realtek RTL8002 and RTL8012 chips) |
| bagetlance.c | Ethernet driver for VME Lance cards on Baget/MIPS |
| declance.c | Lance ethernet driver for the MIPS processor based DECstation family |
| depca.c | A DIGITAL DEPCA & EtherWORKS ethernet driver for Linux |
| eepro.c | Intel EtherExpress Pro/10 device driver for Linux |
| eexpress.c | Intel EtherExpress 16 device driver for Linux |
| epic100.c | A SMC 83c170 EPIC/100 Fast Ethernet driver for Linux (This driver is for the SMC83c170/175 “EPIC” series, as used on the SMC EtherPower II 9432 PCI adapter, and several CardBus cards) |
| eth16i.c | An ICL EtherTeam 16i and 32 EISA Ethernet driver for Linux |
| fmv18x.c | A network device driver for the Fujitsu FMV-181/182/183/184 |
| gmac.c | Network device driver for the GMAC Ethernet controller on Apple G4 Powermacs |
| isa-skeleton.c | A network driver outline for Linux |
| lance.c | An AMD LANCE/PCnet Ethernet driver for Linux |
| lasi_82596.c | Driver for the Intel 82596 Ethernet controller, as munged into HPPA boxen |
| lp486e.c | Panther 82596 Ethernet driver for Linux |
| ni5010.c | A network driver for the MiCom-Interlan NI5010 ethercard |
| ni52.c | net-3-driver for the NI5210 card (i82586 Ethernet chip) |
| ni65.c | ni6510 (am7990 'lance' chip) driver for Linux-net-3 |
| pci-skeleton.c | This driver is for boards based on the RTL8129 and RTL8139 PCI Ethernet chips |
| saa9730.c | SAA9730 Ethernet driver |
| seeq8005.c | A network device driver for the SEEQ 8005 chipset |
| sgiseeq.c | Seeq8003 Ethernet driver for SGI machines |
| sk_g16.c | |
| smc9194.c | A driver for SMC's 9000 series of Ethernet cards |
| sonic.c | |
| sun3lance.c | |

| Device Driver | Original Description |
|---|---|
| tc35815.c | |
| via-rhine.c | A Linux Ethernet device driver for VIA Rhine family chips |
| wavelan.c | WaveLAN ISA driver |
| znet.c | An Zenith Z-Note Ethernet driver for Linux |

Table A.1: Some Vulnerable Linux Device Drivers

The following table lists some vulnerable PCMCIA device drivers from the Linux 2.4.18 Kernel which can be found under `/usr/src/linux/drivers/net/pcmcia`:

| Device Driver | Original Description |
|---|---|
| Wavelan_cs.c | Supports version 2.00 of WaveLAN/PCMCIA cards (2.4GHz) |
| xirc2ps_cs.c | Xircom CreditCard Ethernet Adapter IIps driver<br>Xircom Realport 10/100 (RE-100) driver.<br><br>This driver supports various Xircom CreditCard Ethernet adapters including the CE2, CE IIps, RE-10, CEM28, CEM33, CE33, CEM56, CE3-100, CE3B, RE-100, REM10BT, and REM56G-100. |

Table A.2: Some Vulnerable PCMCIA Linux Device Drivers