CITI Technical Report 99-3

Webcard: a Java Card web server

Jim Rees
Peter Honeyman

info@citi.umich.edu

ABSTRACT

Webcard is a TCP/IP stack and web server written in Java that runs on a Schlumberger Cyberflex Access smartcard. In this report, we describe the architecture and implementation of Webcard and the constraints and assumptions that influenced its design. We also include complete sources for the application and its supporting environment.

October 1, 1999

Center for Information Technology Integration
University of Michigan
519 W. William St.
Ann Arbor, MI 48103-4943

Webcard: a Java Card web server

Jim Rees
Peter Honeyman

info@citi.umich.edu

1. Introduction

The Program for Smartcard Technology at the University of Michigan's Center for Information Technology Integration (CITI) is a research partnership with Schlumberger's Austin Product Center. The Program is actively engaged in research projects that enhance and extend the capabilities of smartcards. Among CITI's goals in the Program, two stand out:

- innovative computer security applications of smartcards, and
- new models of interaction with smartcards.

In this report, we describe Webcard, a web server that is entirely contained in a commercial, off-the-shelf smartcard.

Webcard accomplishes both of CITI's objectives in the categories of research stated above. Webcard takes advantage of the inherent security properties of smartcards, such as tamper resistance and a programming interface appropriate for security applications. While smartcards have traditionally suffered from arcane, operating system dependent applications, Webcard also offers a radically new mode of interacting with smartcards, one that is enabled by any Internet-capable web browser.

2. Technical details

Webcard is a web server running on a Schlumberger Cyberflex Access Java Card [Cyberflex]. The card is programmed by the manufacturer to implement a Java virtual machine (JVM), recognizing a sizable subset of the Java programming language. Specifically, Cyberflex implements the Java Card 2.0 specification [JavaCard]. Java Card is intended to support multiple applications on a single card, as described in ISO 7816-4 [ISO 7816] and EMV 96 [EMV 96]. Webcard is written as a single Java Card application (variously called an *applet* or *cardlet*).

The Cyberflex Access card has 16 KB of EEPROM and about 1.2 Kbytes of RAM. These limited resources make it very difficult to implement a full, standards-compliant version of TCP/IP [RFC 791, RFC 793]. While that is our ultimate goal, we must also accommodate the size limitations imposed by current smartcards; we find it useful and interesting to see how much we can accomplish in as little space as possible.

As a first step toward implementing a standards compliant TCP/IP stack, we elected to implement a minimal, functional server. Our main "robustness" criterion is to produce a server that responds to valid inputs and does not crash when presented with invalid inputs. We depend on the TCP peer to assure reliable operation.

HTTP [RFC 1945], TCP, and IP specify many requirements, many of which are rarely or never used in practice. For our first implementation, we elected to leave out those specifications that are not required in normal operation. To determine which parts of the protocol are actually used, we captured `tcpdump` traces of HTTP transactions from several different clients against an existing server. The assumptions described below are based on the observed traces.

2.1. One Connection at a Time

The Webcard server is simplified by making the assumption that only one connection is active at any time. This allows the server to preserve state for a single connection until a new request comes in. This also eliminates the need to time out defunct connections and to respond to most state change requests. However, most web browsers run requests in parallel, so the server must not return pages with inline content such as images.

It should not be difficult to relax this restriction. The only connection state kept by the Webcard is the file name; TCP state, which is remembered but never used; and TCP port, to enforce the one connection restriction. Connections can be discarded in LRU order as new connection requests arrive, eliminating the need for a timer, which is unavailable on the Cyberflex Access platform.

2.2. HTTP

The server speaks a subset of the HTTP 1.0 protocol, which is simpler and easier to implement than HTTP 1.1 or later. Earlier versions of HTTP, such as HTTP 0.9, are unable to communicate with Webcard, but these clients are now very rare. Modern web clients implement HTTP 1.1 or later, which are required to be backward compatible with HTTP 1.0.

Each request is handled as an individual TCP connection. The HTTP status line, “HTTP/1.0 200 OK”, and the HTTP headers are stored in the files being served, so the server itself does not generate any headers or send any data other than what is in the file.

An HTTP 1.0 GET request consists of the string “GET” followed by a single space character, followed by a server-relative URL. (Webcard does not support any other methods, such as HEAD, POST, or PUT.) For now, URLs are assumed to be three characters, with the last two characters being the file name. (ISO 7816-4 file names are two bytes.)

When the server receives a request, it selects the requested file. It does not store any other state that reflects the identity of the requested file. This implies that only a single HTTP connection can be active at any time, as described above.

2.3. TCP

The server has no configuration information. The network connection is point-to-point, so all incoming packets are assumed to be addressed to the server. The TCP stack simply swaps the source and destination addresses when it constructs a reply packet. No subnet or routing information is required.

Webcard discards any packets not addressed to the HTTP port (TCP port 80). Any TCP options are ignored.

The TCP state machine only has three states: LISTEN, ESTABLISHED, and FIN-WAIT-1. It is incapable of initiating a connection, and does not have the corresponding SYN-SENT state. It also does not have a CLOSED state. Although Webcard keeps track of the TCP state, it makes no use of this information.

The TCP stack never retransmits. This eliminates the need for timers, which are unavailable anyway, and for keeping track of (most) TCP state. We assume the TCP peer retransmits when necessary. In practice, packets are rarely dropped.

The state machine responds to four types of packets. A SYN elicits a SYN ACK reply and transitions to ESTABLISHED, without waiting for the peer to ACK the SYN. We assume that the SYN ACK will not be dropped and will eventually arrive. This assumption is benign: if SYN ACK does get dropped, the peer will retransmit the SYN, allowing connection establishment to proceed.

HTTP 1.0 allows only one line of text to be sent to the server; following our restrictions to HTTP 1.0 described above, any packet with data is assumed to be a complete HTTP GET request. Webcard URLs are exactly three bytes. We assume that the seven bytes in a GET URL request arrive in a single, unfragmented TCP segment. The server extracts the URL from this request and selects the given file in the ISO 7816-4 file system. If the file does not exist, the server selects a file named “nf”, which contains a “404 Not Found” error message. The data packet elicits an ACK of the client's sequence number.

A FIN elicits an ACK and transitions the TCP state machine to LISTEN. HTTP clients always wait for the server to close the connection, so there is no CLOSE-WAIT or LAST-ACK state. If the client does try to close the connection prematurely, it will wait in vain for FIN from the Webcard and will be stuck in FIN-WAIT-2 indefinitely. Most TCP clients eventually recover from this.

An ACK with no data attached elicits data from the currently selected file. There is no windowing -- data is sent when the ACK for the previous segment arrives. Webcard sequence numbers always start at zero, so the client's ACK number gives the offset into the file.

Webcard does not check the client's checksum and ignores the offered window, urgent flag and pointer, and push flag. RST packets are ignored. Outgoing packets always offer a small fixed window. The actual size of this window is unimportant -- we assume the client will never want to send more than 17 bytes.

2.4. IP

Incoming packets are assumed to contain no IP options. It would not be difficult to ignore options, but in practice IP options are never used. The IP header checksum must be done with 16 bit arithmetic because the card does not implement 32 bit arithmetic, but the checksum routine can be simplified by noting that an IP header is never long enough to overflow a 16 bit sum.

The MRU (incoming MTU) is limited by the ISO interface to slightly less than 256 bytes. Webcard does not implement IP reassembly, because the only important incoming information is the URL, which fits in the first 17 bytes.

2.5. Cardlet details

Cyberflex extends Java Card in a number of ways. Cyberflex cardlets contain a main method in addition to the Java Card methods. This allows them to function as standalone programs, but Webcard does not depend on this feature.

A cardlet must have at least three methods, "install," "select," and "process." The install method is invoked once at the time the card is initialized. It creates and initializes the objects needed by the applet. The select method is invoked at the time the cardlet is selected, usually via the "select" application protocol data unit (or *APDU*). A cardlet can be set as the default for the card, in which case that cardlet is implicitly selected whenever the card is used.

The process method does all the work. When an APDU is sent to the card, that APDU is passed to the process method of the currently selected cardlet. IP packets are sent to the Webcard encapsulated in an APDU that gets passed to the process method.

On reset, the Cyberflex Access default loader waits for an incoming APDU and passes it to the ip7816 cardlet. If the APDU is an IP packet (INS=0x12), the cardlet processes the APDU; otherwise the cardlet passes the APDU back to the default loader.

The Webcard cardlet extracts the data length, destination port, and several other fields from the IP and TCP headers, then enters the TCP state machine. It then constructs a reply packet if needed, optionally attaches outgoing data to it, computes TCP and IP checksums, and sends the reply packet as outgoing 7816 data.

At several points in this process the cardlet calls `apdu.waitExtension()` to send a 7816 no-op to the card terminal. This prevents the terminal from timing out while the card is processing.

The Webcard cardlet is about 1200 bytes of Java byte code, leaving about 14 Kbytes of space for web content.

2.6. Card Management

Content is loaded onto the Webcard using SCFS [Itoi], CITI's extension to the UNIX operating system, which mounts any ISO 7816-4 smartcard file system into the UNIX file system name space.

Cardlets can be written in any Java development environment; we tend to use standard UNIX editors and Sun Microsystem's JDK [JDK] for compiling into byte code. A Cyberflex-specific tool called MakeSolo converts the class file into a cardlet ready for downloading with another tool from the Cyberflex developers kit.

2.7. Host Interface

The Cyberflex Access card includes an ISO 7816-3 interface. We use this framing protocol instead of implementing a more conventional serial protocol such as SLIP or PPP. IP packets are encapsulated in a 7816 APDU, with no additional headers. The maximum size of an APDU is 256 bytes. A simple daemon running on OpenBSD (or potentially any system with a tunnel device) forwards packets to the card. The daemon does not implement IP fragmentation, and truncates any packet too big to fit in an APDU. The source code for the OpenBSD tunnel device is included in an Appendix.

Each incoming packet results in at most one reply packet. Cyberflex Access supports 7816-3 T=0 protocol, so the reply packet is retrieved by the daemon with a "get response" APDU.

Routing packets to the Webcard requires external advertisement of the existence of the tunnel. At CITI, we assign the Webcard an otherwise unused IP address from the local subnet's address space and install a static route on our upstream router. On the host to which the card reader is attached, we configure with the following commands:

```
# configure the tunnel
ifconfig tun0 141.211.169.2 smarty
# route through the tunnel
route add smarty 141.211.169.2
# start the tunnel daemon
ip7816d 141.211.169.2
```

2.8. Physical Characteristics

The physical dimensions of Webcard correspond to ISO 7810 ID-1: 85.6 x 54 x .76 mm. Of this, roughly 10 x 12 mm is chip carrier. The chip itself is less than 25 square mm. in size.

3. Discussion

Webcard performance is less than spectacular: approximately 130 bytes per second. We believe this can be accounted for in the main by code path through the JVM. We plan to address performance issues when we are satisfied with functionality.

We intend to extend the functionality of Webcard in many directions, but are mostly concerned with providing better HTTP, TCP, and IP compliance. Our first priority is to address "hosts requirements" such as ICMP functionality, which proves useful in remotely diagnosing problems with IP.

With a more functional TCP/IP stack in hand, we plan to investigate the potential of remote method invocations from host applications. We are also interested in investigating IPv6 and mobile IP for the flexibility they offer to the highly mobile computers embedded in smartcards.

Acknowledgements

We thank Scott Guthery and Tim Jurgensen for valuable advice and suggestions. This work was partially supported by Schlumberger, Inc.

References

[Cyberflex] Schlumberger, Inc., "Cyberflex Access Programmer's Guide" (1998).

[EMV 96] Europay International S.A., MasterCard International Inc., and Visa International Service Assoc., "EMV '96 – Integrated Circuit Card Specification for Payment Systems" (May 1998).

[ISO7816] International Organization for Standardization, "International Standard ISO/IEC 7816: Integrated circuit(s) cards with contacts."

[Itoi] N. Itoi, P. Honeyman, and J. Rees, "SCFS: A UNIX Filesystem for Smartcards," *in Proc. USENIX Workshop on Smartcard Technology*, Chicago (May 1999).

[JavaCard] Sun Microsystems, "Java Card 2.0 Programming Concepts" (October 1997).

[JDK] Sun Microsystems, "Java Card Applet Developer's Guide" (July 1998).

[RFC 791] J. Postel (ed.), "Internet Protocol – DARPA Internet Program Protocol Specification," USC Information Sciences Institute (September 1981).

[RFC 793] J. Postel (ed.), "Transmission Control Protocol – DARPA Internet Program Protocol Specification," USC Information Sciences Institute (September 1981).

[RFC 1945] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol – HTTP/1.0," USC Information Sciences Institute (May 1996).

Appendix: Webcard sources

```
// Copyright (c) 1999
// The Regents of The University of Michigan
// All rights reserved

// Permission is granted to use, copy and redistribute this software
// for noncommercial education and research purposes, so long as no
// fee is charged, and so long as the copyright notice above, this
// grant of permission, and the disclaimer below appear in all copies
// made; and so long as the name of The University of Michigan is not
// used in any advertising or publicity pertaining to the use or
// distribution of this software without specific, written prior
// authorization.  Permission to modify or otherwise create derivative
// works of this software is not granted.
//
// This software is provided as is, without representation as to its
// fitness for any purpose, and without warranty of any kind, either
// express or implied, including without limitation the implied
// warranties of merchantability and fitness for a particular purpose.
// The Regents of The University of Michigan shall not be liable for
// any damages, including special, indirect, incidental, or consequential
// damages, with respect to any claim arising out of or in connection
// with the use of the software, even if it has been or is hereafter
// advised of the possibility of such damages.

// Contact: info@citi.umich.edu


// A very small tcp stack and web server for Schlumberger Cyberflex Access

import javacard.framework.*;
import javacardx.framework.*;

public class ip7816 extends Applet
{
    static final byte CMD_IP = (byte)0x12;

    static final byte FL_ACK = 0x10;
    static final byte FL_PSH = 0x8;
    static final byte FL_RST = 0x4;
    static final byte FL_SYN = 0x2;
    static final byte FL_FIN = 0x1;

    static final byte ST_LISTEN = 0;
    static final byte ST_ESTAB = 2;
    static final byte ST_FW1 = 3;
    static final byte ST_FW2 = 4;

    static final short CD = ISO.OFFSET_CDATA;
    static final short MTU = 248; // Fits in a 256 byte apdu

    // "TCB"
    // In a full tcp implementation we would keep track of this per connection.
    // This implementation only handles one connection at a time.
    // As a result, very little of this state is actually used after
    // the reply packet has been sent.

    byte src[], dst[];
    short id, tcb_port;    // ip id, tcp port
    byte tcb_st;    // state
```

```
private ip7816() {
   src = new byte[4];
   dst = new byte[4];
   register();
}

public static void install(APDU apdu) {
   new ip7816();
}

public boolean select() {
   id = 1;
   tcb_st = ST_LISTEN;
   return true;
}

public static void main(String args[]) {
   ISOException.throwIt((short) 0x811F);
}

public void process(APDU apdu) {
   short i, len, port, rcvnxt0, rcvnxt1, sndnxt, offset, datlen, ck, d0, d1;
   byte pkt[] = apdu.getBuffer(), fl;

   switch (pkt[ISO.OFFSET_INS]) {

   case CMD_IP:
       // Incoming IP packet
       len = apdu.setIncomingAndReceive();
       if (len < 40)
          ISOException.throwIt(ISO.SW_WRONG_LENGTH);
       // Packet may have been truncated by ip7816d; find real len
       len = Util.getShort(pkt, (short)(CD+2));

       // If it’s not http, just drop it
       if (Util.getShort(pkt, (short)(CD+22)) != 80)
          break;

       // Get source and destination address and source port
       Util.arrayCopy(pkt, (short)(CD+12), src, (short)0, (short)4);
       Util.arrayCopy(pkt, (short)(CD+16), dst, (short)0, (short)4);
       port = Util.getShort(pkt, (short)(CD+20));

       // Get the sender’s sequence and ack
       // XXX Note this is 16-bit; we don’t handle overflow
       rcvnxt0 = Util.getShort(pkt, (short)(CD+24));
       rcvnxt1 = Util.getShort(pkt, (short)(CD+26));
       sndnxt = Util.getShort(pkt, (short)(CD+30));

       // Find the payload
       offset = (short) (((pkt[CD+32] >> 2) & 0x3c) + 20);
       datlen = (short) (len - offset);

       len = 40;
       fl = FL_ACK;

       apdu.waitExtension();
```

```
// Figure out what kind of packet this is, and respond
if ((pkt[CD+33] & FL_SYN) != 0) {
   // SYN
   sndnxt = 0;
   rcvnxt1++;
   fl |= FL_SYN;
   tcb_st = ST_ESTAB;

} else if (datlen > 0) {

   // incoming data
   rcvnxt1 += datlen;

   // Get the url (two chars after "GET /")
   if (pkt[CD+offset+5] == 0x20)
       // Turn "/" into "in"
       d0 = 0x696e;
   else
       d0 = Util.getShort(pkt, (short)(CD+offset+5));

   // select the file and get its size
   // if file not found, try "nf" then "in"
   if (CyberflexFile.selectFile(d0) != ST.SUCCESS
       && CyberflexFile.selectFile((short)0x6e66) != ST.SUCCESS)
       CyberflexFile.selectFile((short)0x696e);
   len += (short) (CyberflexFile.getFileSize() - 16);
   fl |= FL_PSH;
   tcb_port = port;

} else if ((pkt[CD+33] & FL_FIN) != 0) {

   // FIN
   rcvnxt1++;
   // Don't bother with FIN-WAIT-2, TIME-WAIT, or CLOSED
   tcb_st = ST_LISTEN;

} else if ((pkt[CD+33] & FL_ACK) != 0) {

   // ack with no data
   if (tcb_port == port && sndnxt > 1) {
       // calculate no of bytes left to send
       i = (short) (CyberflexFile.getFileSize() - 16 - (sndnxt - 1));
       if (i == 0) {
          // EOF; send FIN
          fl |= FL_FIN;
          tcb_st = ST_FW1;
       } else if (i > 0) {
          // not EOF; send next segment
          len += i;
          fl |= FL_PSH;
       } else {
          // ack of FIN; no reply
          break;
       }
   } else
       break;   // No reply packet

} else

   break; // drop it

apdu.waitExtension();
```

```
// Send reply packet
if (len > MTU)
   len = MTU;

// Read next segment of data into buffer
if (len > 40)
   CyberflexOS.readBinaryFile(pkt, (short)40, (short)(sndnxt - 1),
                                              (short)(len - 40));

apdu.waitExtension();

for (i = 0; i < 40; i++)
   pkt[i] = 0;

// Fill in IP header
pkt[0] = 0x45;  // version, header len
Util.setShort(pkt, (short)2, len);
Util.setShort(pkt, (short)4, id);
pkt[8] = 60;    // ttl
pkt[9] = 6;           // protocol (tcp)
Util.arrayCopy(dst, (short)0, pkt, (short)12, (short)4);
Util.arrayCopy(src, (short)0, pkt, (short)16, (short)4);

apdu.waitExtension();

// Calculate IP header checksum
ck = d0 = d1 = 0;
for (i = 0; i < 20; i += 2) {
   d0 += (short) (pkt[i] & 0xff);
   d1 += (short) (pkt[i+1] & 0xff);
}
// This works because IP header is too short to overflow high byte
ck = (short) ~(((d0 >> 8) & 0xff) + (d0 << 8) + d1);
pkt[10] = (byte) (ck >> 8);
pkt[11] = (byte) ck;

apdu.waitExtension();

// Fill in TCP header
pkt[21] = 80;   // Source port
Util.setShort(pkt, (short)22, port);
Util.setShort(pkt, (short)26, sndnxt);
Util.setShort(pkt, (short)28, rcvnxt0);
Util.setShort(pkt, (short)30, rcvnxt1);
pkt[32] = 0x50; // data offset = 20 (no options)
pkt[33] = fl;   // flags
pkt[34] = 0x0a; // window = 2680
pkt[35] = 0x78;

apdu.waitExtension();

// Calculate TCP checksum
ck = d0 = d1 = 0;
pkt[len] = 0;
for (i = 12; i < len; i += 2) {
   d0 += (short) (pkt[i] & 0xff);
   d1 += (short) (pkt[i+1] & 0xff);
}
```

```
            d1 += 6 + len - 20;
            ck = (short) ((d0 & 0xff) + ((d1 >> 8) & 0xff));
            ck = (short) ~(((d0 >> 8) & 0xff) + (d1 & 0xff)
                           + ((ck >> 8) & 0xff) + (ck << 8));
            pkt[36] = (byte) (ck >> 8);
            pkt[37] = (byte) ck;

            // Send return packet
            apdu.setOutgoingAndSend((short)0, len);

            break;

        }
    }
}
```

Appendix: Tunnel daemon sources

```
/*
Copyright (c) 1999
The Regents of The University of Michigan
All rights reserved

Permission is granted to use, copy and redistribute this software
for noncommercial education and research purposes, so long as no
fee is charged, and so long as the copyright notice above, this
grant of permission, and the disclaimer below appear in all copies
made; and so long as the name of The University of Michigan is not
used in any advertising or publicity pertaining to the use or
distribution of this software without specific, written prior
authorization.  Permission to modify or otherwise create derivative
works of this software is not granted.

This software is provided as is, without representation as to its
fitness for any purpose, and without warranty of any kind, either
express or implied, including without limitation the implied
warranties of merchantability and fitness for a particular purpose.
The Regents of The University of Michigan shall not be liable for
any damages, including special, indirect, incidental, or consequential
damages, with respect to any claim arising out of or in connection
with the use of the software, even if it has been or is hereafter
advised of the possibility of such damages.

Contact: info@citi.umich.edu
*/

/*
 * Read packets from tunnel device and send them to a smartcard
 *
 * Command line options:
 * -[12]  serial port to use, 1 is default
 * -l     log (to stdout) incoming connection requests
 * -v     log (to stdout) all apdus (but not their contents)
 */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "scrw.h"

#define IP_CLA 0x4
#define IP_INS 0x12
#define GR_INS 0xc0

char tunneldevname[] = "/dev/tun0";

int vflag, lflag;

main(ac, av)
int ac;
char *av[];
{
    extern int optind;
    int scfd, ipfd, port = 0, i, d0, n, af, len, r1, r2;
    char *s, buf[100];
    static unsigned char pkt[1504];
```

```
while ((i = getopt(ac, av, "12lv")) != -1) {
   switch (i) {
   case ’1’:
   case ’2’:
       port = i - ’1’;
       break;
   case ’l’:
       lflag = 1;
       break;
   case ’v’:
       vflag = 1;
       break;
   }
}

setlinebuf(stdout);

/* open reader and reset */
scfd = scopen(port, SCODSR, NULL);
if (scfd < 0) {
   printf("can’t open reader\n");
   exit(1);
}
screset(scfd, NULL, NULL);

/* open tunnel device */
if (optind < ac) {
   sprintf(buf, "/sbin/ifconfig tun0 %s", av[optind]);
   system(buf);
}
ipfd = open(tunneldevname, 2);
if (ipfd < 0) {
   perror(tunneldevname);
   exit(1);
}
/* why is this necessary? */
system("/sbin/ifconfig tun0 up");

while (1) {
   n = read(ipfd, pkt, sizeof pkt);
   if (n <= 0) {
       printf("eof\n");
       break;
   }
   if (n < 44) {
       printf("short packet %d\n", n);
       continue;
   }

   /* Save the address family, move the rest of the packet up */
   memmove(&af, pkt, 4);
   memmove(pkt, pkt + 4, n - 4);

   if (pkt[0] != 0x45) {
       printf("bad version or IHL %02x\n", pkt[0]);
       continue;
   }
   len = (pkt[2] << 8) | pkt[3];
   if (len < 40 || len > 1500 || len != n - 4) {
       printf("bad len %02x\n", len);
       continue;
   }
```

```
        if (lflag && pkt[9] == 6 && (pkt[33] & 0x2))
            printf("SYN from %d.%d.%d.%d\n", pkt[12], pkt[13], pkt[14], pkt[15]);

        if (len > 80)
            len = 80;

        scwrite(scfd, IP_CLA, IP_INS, 0, 0, len, pkt, &r1, &r2);

        if (vflag) {
            s = scr1r2s(r1, r2);
            if (!s)
               s = "";
            printf("sent pkt len %d status %02x.%02x %s\n", len, r1, r2, s);
        }

        if (r1 == 0x90)
            /* No return packet */
            continue;

        if (r1 != 0x61) {
            s = scr1r2s(r1, r2);
            if (!s)
               s = "";
            printf("get_response(0) status %02x.%02x %s\n", r1, r2, s);
            continue;
        }
        len = r2;
        if (len == 0)
            continue;

        /* Read the return packet */
        n = scread(scfd, IP_CLA, GR_INS, 0, 0, len, pkt, &r1, &r2);
        if (n != len) {
            printf("bad len wanted %d got %d\n", len, n);
            len = n;
        }
        if (vflag) {
            s = scr1r2s(r1, r2);
            if (!s)
               s = "";
            printf("rcvd pkt len %d status %02x.%02x %s\n", len, r1, r2, s);
        }

        /* Insert address family and write to tunnel */
        memmove(pkt + 4, pkt, len);
        memmove(pkt, &af, 4);
        write(ipfd, pkt, len + 4);
    }

    scclose(scfd);
    close(ipfd);
    exit(0);
}
```