```
**********************************************************
   51481 Tue Dec  2 07:34:51 2014
new/usr/src/cmd/cmd-inet/usr.bin/pppd/options.c
5378 CVE-2014-3158 ppp: integer overflow in option parsing
Reviewed by: Robert Mustacchi <rm@joyent.com>
**********************************************************
```

```
/*
 * options.c - handles option processing for PPP.
 *
 * Copyright (c) 2000-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation is hereby granted, provided that the above copyright
 * notice appears in all copies.
 *
 * SUN MAKES NO REPRESENTATION OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT.  SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES
 *
 * Copyright (c) 1989 Carnegie Mellon University.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by Carnegie Mellon University.  The name of the
 * University may not be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"
#define RCSID   "$Id: options.c,v 1.74 2000/04/15 01:27:13 masputra Exp $"

#include <ctype.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <syslog.h>
#include <string.h>
#include <netdb.h>
#include <pwd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#ifdef PLUGIN
#include <dlfcn.h>
#endif /* PLUGIN */
#ifdef PPP_FILTER
#include <pcap.h>
#include <pcap-int.h>   /* XXX: To get struct pcap */
#endif /* PPP_FILTER */

#include "pppd.h"
#include "pathnames.h"
```



```
#include "patchlevel.h"
#include "fsm.h"
#include "lcp.h"
#include "ipcp.h"

#if defined(ultrix) || defined(NeXT)
char *strdup __P((char *));
#endif

#if !defined(lint) && !defined(_lint)
static const char rcsid[] = RCSID;
#endif

/*
 * Option variables and default values.
 */
#ifdef PPP_FILTER
int     dflag = 0;              /* Tell libpcap we want debugging */
#endif /* PPP_FILTER */
int     debug = 0;              /* Debug flag */
int     kdebugflag = 0;         /* Tell kernel to print debug messages */
int     default_device = 1;     /* Using /dev/tty or equivalent */
char    devnam[MAXPATHLEN];     /* Device name */
int     crtscts = 0;            /* Use hardware flow control */
bool    modem = 1;              /* Use modem control lines */
int     inspeed = 0;            /* Input/Output speed requested */
u_int32_t netmask = 0;          /* IP netmask to set on interface */
bool    lockflag = 0;           /* Create lock file to lock the serial dev */
bool    nodetach = 0;           /* Don't detach from controlling tty */
bool    updetach = 0;           /* Detach once link is up */
char    *initializer = NULL;    /* Script to initialize physical link */
char    *connect_script = NULL; /* Script to establish physical link */
char    *disconnect_script = NULL; /* Script to disestablish physical link */
char    *welcomer = NULL;       /* Script to run after phys link estab. */
char    *ptycommand = NULL;     /* Command to run on other side of pty */
int     maxconnect = 0;         /* Maximum connect time */
char    user[MAXNAMELEN];       /* Username for PAP */
char    passwd[MAXSECRETLEN];   /* Password for PAP */
bool    persist = 0;            /* Reopen link after it goes down */
char    our_name[MAXNAMELEN];   /* Our name for authentication purposes */
bool    demand = 0;             /* do dial-on-demand */
char    *ipparam = NULL;        /* Extra parameter for ip up/down scripts */
int     idle_time_limit = 0;    /* Disconnect if idle for this many seconds */
int     holdoff = 30;           /* # seconds to pause before reconnecting */
bool    holdoff_specified;      /* true if a holdoff value has been given */
bool    notty = 0;              /* Stdin/out is not a tty */
char    *pty_socket = NULL;     /* Socket to connect to pty */
char    *record_file = NULL;    /* File to record chars sent/received */
int     using_pty = 0;
bool    sync_serial = 0;        /* Device is synchronous serial device */
int     log_to_fd = 1;          /* send log messages to this fd too */
int     maxfail = 10;           /* max # of unsuccessful connection attempts */
char    linkname[MAXPATHLEN];   /* logical name for link */
bool    tune_kernel;            /* may alter kernel settings */
int     connect_delay = 1000;   /* wait this many ms after connect script */
int     max_data_rate;          /* max bytes/sec through charshunt */
int     req_unit = -1;          /* requested interface unit */
bool    multilink = 0;          /* Enable multilink operation */
char    *bundle_name = NULL;    /* bundle name for multilink */
bool    direct_tty = 0;         /* use standard input directly; not a tty */

/* Maximum depth of include files; prevents looping. */
#define MAXFILENESTING  10

struct option_info initializer_info;
struct option_info connect_script_info;
```

```
 126 struct option_info disconnect_script_info;
 127 struct option_info welcomer_info;
 128 struct option_info devnam_info;
 129 struct option_info ptycommand_info;
 130 struct option_info ipsrc_info;
 131 struct option_info ipdst_info;
 132 struct option_info speed_info;

 134 #ifdef PPP_FILTER
 135 struct  bpf_program pass_filter;/* Filter program for packets to pass */
 136 struct  bpf_program active_filter; /* Filter program for link-active pkts */
 137 pcap_t  pc;                     /* Fake struct pcap so we can compile expr */
 138 #endif /* PPP_FILTER */

 140 char *current_option;           /* the name of the option being parsed */
 141 bool privileged_option;         /* set iff the current option came from root */
 142 char *option_source = NULL;     /* string saying where the option came from */
 143 int option_line = 0;            /* line number in file */
 144 bool log_to_file;               /* log_to_fd is a file opened by us */
 145 bool log_to_specific_fd;        /* log_to_fd was specified by user option */

 147 /*
 148  * Prototypes.
 149  */
 150 static int setdevname __P((char *));
 151 static int setipaddr __P((char *));
 152 static int setspeed __P((char *));
 153 static int noopt __P((char **, option_t *));
 154 static int setdomain __P((char **, option_t *));
 155 static int setnetmask __P((char **, option_t *));
 156 static int setxonxoff __P((char **, option_t *));
 157 static int readfile __P((char **, option_t *));
 158 static int callfile __P((char **, option_t *));
 159 static int showversion __P((char **, option_t *));
 160 static int showhelp __P((char **, option_t *));
 161 static int showalloptions __P((char **, option_t *));
 162 static void usage __P((void));
 163 static int setlogfile __P((char **, option_t *));
 164 #ifdef PLUGIN
 165 static int loadplugin __P((char **, option_t *));
 166 #endif
 167 #ifdef PPP_FILTER
 168 static int setpassfilter __P((char **, option_t *));
 169 static int setactivefilter __P((char **, option_t *));
 170 #endif /* PPP_FILTER */
 171 static option_t *find_option __P((char *name));
 172 static int process_option __P((option_t *opt, char **argv, int sline));
 173 static int n_arguments __P((option_t *opt));
 174 static int number_option __P((char *str, u_int32_t *valp, int base));
 175 static u_int32_t opt_hash __P((const void *key));
 176 static int opt_compare __P((const void *p1, const void *p2));

 178 typedef struct _opt_t {
 179     option_t    *p;
 180 } opt_t;
_____unchanged_portion_omitted_

1146 /*
1147  * getword - read a word from a file.  Words are delimited by white-space or by
1148  * quotes (" or ').  Quotes, white-space and \ may be escaped with \.
1149  * \<newline> is ignored.  Returns 1 upon successful processing of options,
1150  * and 0 otherwise.
1151  */
1152 int
1153 getword(f, word, newlinep, filename)
1154     FILE *f;
```



```
1155     char *word;
1156     int *newlinep;
1157     char *filename;
1158 {
1159     int c, len, escape;
1160     int quoted, comment;
1161     int value, digit, got, n;

1163 #define isoctal(c) ((c) >= '0' && (c) < '8')

1165     *newlinep = 0;
1166     len = 0;
1167     escape = 0;
1168     comment = 0;

1170     /*
1171      * First skip white-space and comments.
1172      */
1173     for (;;) {
1174         c = getc(f);
1175         if (c == EOF)
1176             break;

1178         /*
1179          * A newline means the end of a comment; backslash-newline
1180          * is ignored.  Note that we cannot have escape && comment.
1181          */
1182         if (c == '\n') {
1183             option_line++;
1184             if (!escape) {
1185                 *newlinep = 1;
1186                 comment = 0;
1187             } else
1188                 escape = 0;
1189             continue;
1190         }

1192         /*
1193          * Ignore characters other than newline in a comment.
1194          */
1195         if (comment)
1196             continue;

1198         /*
1199          * If this character is escaped, we have a word start.
1200          */
1201         if (escape)
1202             break;

1204         /*
1205          * If this is the escape character, look at the next character.
1206          */
1207         if (c == '\\') {
1208             escape = 1;
1209             continue;
1210         }

1212         /*
1213          * If this is the start of a comment, ignore the rest of the line.
1214          */
1215         if (c == '#') {
1216             comment = 1;
1217             continue;
1218         }

1220         /*
```

```
1221          * A non-whitespace character is the start of a word.
1222          */
1223         if (!isspace(c))
1224             break;
1225     }
1226
1227     /*
1228      * Save the delimiter for quoted strings.
1229      */
1230     if (!escape && (c == '"' || c == '\'')) {
1231         quoted = c;
1232         c = getc(f);
1233     } else
1234         quoted = 0;
1235
1236     /*
1237      * Process characters until the end of the word.
1238      */
1239     while (c != EOF) {
1240         if (escape) {
1241             /*
1242              * This character is escaped: backslash-newline is ignored,
1243              * various other characters indicate particular values
1244              * as for C backslash-escapes.
1245              */
1246             escape = 0;
1247             if (c == '\n') {
1248                 c = getc(f);
1249                 continue;
1250             }
1251
1252             got = 0;
1253             switch (c) {
1254             case 'a':
1255                 value = '\a';
1256                 break;
1257             case 'b':
1258                 value = '\b';
1259                 break;
1260             case 'f':
1261                 value = '\f';
1262                 break;
1263             case 'n':
1264                 value = '\n';
1265                 break;
1266             case 'r':
1267                 value = '\r';
1268                 break;
1269             case 's':
1270                 value = ' ';
1271                 break;
1272             case 't':
1273                 value = '\t';
1274                 break;
1275
1276             default:
1277                 if (isoctal(c)) {
1278                     /*
1279                      * \ddd octal sequence
1280                      */
1281                     value = 0;
1282                     for (n = 0; n < 3 && isoctal(c); ++n) {
1283                         value = (value << 3) + (c & 07);
1284                         c = getc(f);
1285                     }
1286                     got = 1;
```



```
1287                     break;
1288                 }
1289
1290                 if (c == 'x') {
1291                     /*
1292                      * \x<hex_string> sequence
1293                      */
1294                     value = 0;
1295                     c = getc(f);
1296                     for (n = 0; n < 2 && isxdigit(c); ++n) {
1297                         digit = (islower(c) ? toupper(c) : c) - '0';
1298                         if (digit > 10 || digit < 0)    /* allow non-ASCII */
1299                             digit += '0' + 10 - 'A';
1300                         value = (value << 4) + digit;
1301                         c = getc (f);
1302                     }
1303                     got = 1;
1304                     break;
1305                 }
1306
1307                 /*
1308                  * Otherwise the character stands for itself.
1309                  */
1310                 value = c;
1311                 break;
1312             }
1313
1314             /*
1315              * Store the resulting character for the escape sequence.
1316              */
1317             if (len < MAXWORDLEN) {
1318             if (len < MAXWORDLEN-1)
1318                 word[len] = value;
1319                 ++len;
1320             }
1321 #endif /* ! codereview */
1322
1323             if (!got)
1324                 c = getc(f);
1325             continue;
1326
1327         }
1328
1329         /*
1330          * Not escaped: see if we've reached the end of the word.
1331          */
1332         if (quoted) {
1333             if (c == quoted)
1334                 break;
1335         } else {
1336             if (isspace(c) || c == '#') {
1337                 (void) ungetc (c, f);
1338                 break;
1339             }
1340         }
1341
1342         /*
1343          * Backslash starts an escape sequence.
1344          */
1345         if (c == '\\') {
1346             escape = 1;
1347             c = getc(f);
1348             continue;
1349         }
1350
1351         /*
```

```
1352          * An ordinary character: store it in the word and get another.
1353          */
1354         if (len < MAXWORDLEN) {
1321         if (len < MAXWORDLEN-1)
1355             word[len] = c;
1356             ++len;
1357         }
1358 #endif /* ! codereview */

1360         c = getc(f);
1361     }

1363     /*
1364      * End of the word: check for errors.
1365      */
1366     if (c == EOF) {
1367         if (ferror(f)) {
1368             if (errno == 0)
1369                 errno = EIO;
1370             option_error("Error reading %s: %m", filename);
1371             die(1);
1372         }
1373         /*
1374          * If len is zero, then we didn't find a word before the
1375          * end of the file.
1376          */
1377         if (len == 0)
1378             return (0);
1379     }

1381     /*
1382      * Warn if the word was too long, and append a terminating null.
1383      */
1384     if (len >= MAXWORDLEN) {
1385         option_error("warning: word in file %s too long (%.20s...)",
1386                      filename, word);
1387         len = MAXWORDLEN - 1;
1388     }
1389     word[len] = '\0';

1391     return (1);

1393 #undef isoctal

1395 }

1397 /*
1398  * number_option - parse an unsigned numeric parameter for an option.
1399  * Returns 1 upon successful processing of options, and 0 otherwise.
1400  */
1401 static int
1402 number_option(str, valp, base)
1403     char *str;
1404     u_int32_t *valp;
1405     int base;
1406 {
1407     char *ptr;

1409     *valp = strtoul(str, &ptr, base);
1410     if (ptr == str || *ptr != '\0') {
1411         option_error("invalid numeric parameter '%s' for '%s' option",
1412                      str, current_option);
1413         return (0);
1414     }
1415     return (1);
1416 }
```



```
1418 /*
1419  * save_source - store option source, line, and privilege into an
1420  * option_info structure.
1421  */
1422 void
1423 save_source(info)
1424     struct option_info *info;
1425 {
1426     info->priv = privileged_option;
1427     info->source = option_source;
1428     info->line = option_line;
1429 }

1431 /*
1432  * set_source - set option source, line, and privilege from an
1433  * option_info structure.
1434  */
1435 void
1436 set_source(info)
1437     struct option_info *info;
1438 {
1439     privileged_option = info->priv;
1440     option_source = info->source;
1441     option_line = info->line;
1442 }

1444 /*
1445  * name_source - return string containing option source and line.  Can
1446  * be used as part of an option_error call.
1447  */
1448 const char *
1449 name_source(info)
1450     struct option_info *info;
1451 {
1452     static char buf[MAXPATHLEN];

1454     if (info->source == NULL)
1455         return "none";
1456     if (info->line <= 0)
1457         return info->source;
1458     (void) slprintf(buf, sizeof (buf), "%s:%d", info->source, info->line);
1459     return (const char *)buf;
1460 }

1462 /*
1463  * int_option - like number_option, but valp is int *, the base is assumed to
1464  * be 0, and *valp is not changed if there is an error.  Returns 1 upon
1465  * successful processing of options, and 0 otherwise.
1466  */
1467 int
1468 int_option(str, valp)
1469     char *str;
1470     int *valp;
1471 {
1472     u_int32_t v;

1474     if (!number_option(str, &v, 0))
1475         return (0);
1476     *valp = (int) v;
1477     return (1);
1478 }


1481 /*
1482  * The following procedures parse options.
```

```
 */

/*
 * readfile - take commands from a file.
 */
/*ARGSUSED*/
static int
readfile(argv, opt)
    char **argv;
    option_t *opt;
{
    return (options_from_file(*argv, 1, 1, privileged_option));
}

/*
 * callfile - take commands from /etc/ppp/peers/<name>.  Name may not contain
 * /../, start with / or ../, or end in /.  Returns 1 upon successful
 * processing of options, and 0 otherwise.
 */
/*ARGSUSED*/
static int
callfile(argv, opt)
    char **argv;
    option_t *opt;
{
    char *fname, *arg, *p;
    int l, ok;

    arg = *argv;
    ok = 1;
    if (arg[0] == '/' || arg[0] == '\0')
        ok = 0;
    else {
        for (p = arg; *p != '\0'; ) {
            if (p[0] == '.' && p[1] == '.' && (p[2] == '/' || p[2] == '\0')) {
                ok = 0;
                break;
            }
            while (*p != '/' && *p != '\0')
                ++p;
            if (*p == '/')
                ++p;
        }
    }
    if (!ok) {
        option_error("call option value may not contain .. or start with /");
        return (0);
    }

    l = strlen(arg) + strlen(_PATH_PEERFILES) + 1;
    if ((fname = (char *) malloc(l)) == NULL)
        novm("call file name");
    (void) slprintf(fname, l, "%s%s", _PATH_PEERFILES, arg);

    ok = options_from_file(fname, 1, 1, 1);

    free(fname);
    return (ok);
}

#ifdef PPP_FILTER
/*
 * setpdebug - set libpcap debugging level.  Returns 1 upon successful
 * processing of options, and 0 otherwise.
 */
static int
```



```
setpdebug(argv)
    char **argv;
{
    return (int_option(*argv, &dflag));
}

/*
 * setpassfilter - set the pass filter for packets.  Returns 1 upon successful
 * processing of options, and 0 otherwise.
 */
/*ARGSUSED*/
static int
setpassfilter(argv, opt)
    char **argv;
    option_t *opt;
{
    pc.linktype = DLT_PPP;
    pc.snapshot = PPP_HDRLEN;

    if (pcap_compile(&pc, &pass_filter, *argv, 1, netmask) == 0)
        return (1);
    option_error("error in pass-filter expression: %s\n", pcap_geterr(&pc));
    return (0);
}

/*
 * setactivefilter - set the active filter for packets.  Returns 1 upon
 * successful processing of options, and 0 otherwise.
 */
/*ARGSUSED*/
static int
setactivefilter(argv, opt)
    char **argv;
    option_t *opt;
{
    pc.linktype = DLT_PPP;
    pc.snapshot = PPP_HDRLEN;

    if (pcap_compile(&pc, &active_filter, *argv, 1, netmask) == 0)
        return (1);
    option_error("error in active-filter expression: %s\n", pcap_geterr(&pc));
    return (0);
}
#endif /* PPP_FILTER */

/*
 * noopt - disable all options.  Returns 1 upon successful processing of
 * options, and 0 otherwise.
 */
/*ARGSUSED*/
static int
noopt(argv, opt)
    char **argv;
    option_t *opt;
{
    BZERO((char *) &lcp_wantoptions[0], sizeof (struct lcp_options));
    BZERO((char *) &lcp_allowoptions[0], sizeof (struct lcp_options));
    BZERO((char *) &ipcp_wantoptions[0], sizeof (struct ipcp_options));
    BZERO((char *) &ipcp_allowoptions[0], sizeof (struct ipcp_options));

    return (1);
}

/*
 * setdomain - set domain name to append to hostname.  Returns 1 upon
 * successful processing of options, and 0 otherwise.
```

```
 */
/*ARGSUSED*/
static int
setdomain(argv, opt)
    char **argv;
    option_t *opt;
{
    if (!privileged_option) {
        option_error("using the domain option requires root privilege");
        return (0);
    }
    (void) gethostname(hostname, MAXHOSTNAMELEN+1);
    if (**argv != '\0') {
        if (**argv != '.')
            (void) strncat(hostname, ".", MAXHOSTNAMELEN - strlen(hostname));
        (void) strncat(hostname, *argv, MAXHOSTNAMELEN - strlen(hostname));
    }
    hostname[MAXHOSTNAMELEN] = '\0';
    return (1);
}


/*
 * setspeed - set the speed.  Returns 1 upon successful processing of options,
 * and 0 otherwise.
 */
static int
setspeed(arg)
    char *arg;
{
    char *ptr;
    int spd;

    if (prepass)
        return (1);
    spd = strtol(arg, &ptr, 0);
    if (ptr == arg || *ptr != '\0' || spd <= 0)
        return (0);
    inspeed = spd;
    save_source(&speed_info);
    return (1);
}


/*
 * setdevname - set the device name.  Returns 1 upon successful processing of
 * options, 0 when the device does not exist, and -1 when an error is
 * encountered.
 */
static int
setdevname(cp)
    char *cp;
{
    struct stat statbuf;
    char dev[MAXPATHLEN];

    if (*cp == '\0')
        return (0);

    if (strncmp("/dev/", cp, 5) != 0) {
        (void) strlcpy(dev, "/dev/", sizeof(dev));
        (void) strlcat(dev, cp, sizeof(dev));
        cp = dev;
    }

    /*
```



```
     * Check if there is a character device by this name.
     */
    if (stat(cp, &statbuf) < 0) {
        if (errno == ENOENT) {
            return (0);
        }
        option_error("Couldn't stat '%s': %m", cp);
        return (-1);
    }
    if (!S_ISCHR(statbuf.st_mode)) {
        option_error("'%s' is not a character device", cp);
        return (-1);
    }

    if (phase != PHASE_INITIALIZE) {
        option_error("device name cannot be changed after initialization");
        return (-1);
    } else if (devnam_fixed) {
        option_error("per-tty options file may not specify device name");
        return (-1);
    }

    if (devnam_info.priv && !privileged_option) {
        option_error("device name %s from %s cannot be overridden",
            devnam, name_source(&devnam_info));
        return (-1);
    }

    (void) strlcpy(devnam, cp, sizeof(devnam));
    devstat = statbuf;
    default_device = 0;
    save_source(&devnam_info);

    return (1);
}


/*
 * setipaddr - set the IP address.  Returns 1 upon successful processing of
 * options, 0 when the argument does not contain a ':', and -1 for error.
 */
static int
setipaddr(arg)
    char *arg;
{
    struct hostent *hp;
    char *colon;
    u_int32_t local, remote;
    ipcp_options *wo = &ipcp_wantoptions[0];

    /*
     * IP address pair separated by ":".
     */
    if ((colon = strchr(arg, ':')) == NULL)
        return (0);
    if (prepass)
        return (1);

    /*
     * If colon first character, then no local addr.
     */
    if (colon != arg) {
        *colon = '\0';
        if ((local = inet_addr(arg)) == (u_int32_t) -1) {
            if ((hp = gethostbyname(arg)) == NULL) {
                option_error("unknown host: %s", arg);
```

```
1747                return (-1);
1748            } else {
1749                BCOPY(hp->h_addr, &local, sizeof(local));
1750            }
1751        }
1752        if (bad_ip_adrs(local)) {
1753            option_error("bad local IP address %I", local);
1754            return (-1);
1755        }
1756        if (local != 0) {
1757            save_source(&ipsrc_info);
1758            wo->ouraddr = local;
1759        }
1760        *colon = ':';
1761    }

1763    /*
1764     * If colon last character, then no remote addr.
1765     */
1766    if (*++colon != '\0') {
1767        if ((remote = inet_addr(colon)) == (u_int32_t) -1) {
1768            if ((hp = gethostbyname(colon)) == NULL) {
1769                option_error("unknown host: %s", colon);
1770                return (-1);
1771            } else {
1772                BCOPY(hp->h_addr, &remote, sizeof(remote));
1773                if (remote_name[0] == '\0')
1774                    (void) strlcpy(remote_name, colon, sizeof(remote_name));
1775            }
1776        }
1777        if (bad_ip_adrs(remote)) {
1778            option_error("bad remote IP address %I", remote);
1779            return (-1);
1780        }
1781        if (remote != 0) {
1782            save_source(&ipdst_info);
1783            wo->hisaddr = remote;
1784        }
1785    }

1787    return (1);
1788 }


1791 /*
1792  * setnetmask - set the netmask to be used on the interface.  Returns 1 upon
1793  * successful processing of options, and 0 otherwise.
1794  */
1795 /*ARGSUSED*/
1796 static int
1797 setnetmask(argv, opt)
1798     char **argv;
1799     option_t *opt;
1800 {
1801     u_int32_t mask;
1802     int n;
1803     char *p;

1805     /*
1806      * Unfortunately, if we use inet_addr, we can't tell whether
1807      * a result of all 1s is an error or a valid 255.255.255.255.
1808      */
1809     p = *argv;
1810     n = parse_dotted_ip(p, &mask);

1812     mask = htonl(mask);
```



```
1814     if (n == 0 || p[n] != 0 || (netmask & ~mask) != 0) {
1815         option_error("invalid netmask value '%s'", *argv);
1816         return (0);
1817     }

1819     netmask = mask;
1820     return (1);
1821 }

1823 /*
1824  * parse_dotted_ip - parse and convert the IP address string to make
1825  * sure it conforms to the dotted notation.  Returns the length of
1826  * processed characters upon success, and 0 otherwise.  If successful,
1827  * the converted IP address number is stored in vp, in the host byte
1828  * order.
1829  */
1830 int
1831 parse_dotted_ip(cp, vp)
1832     register char *cp;
1833     u_int32_t *vp;
1834 {
1835     register u_int32_t val, base, n;
1836     register char c;
1837     char *cp0 = cp;
1838     u_char parts[3], *pp = parts;

1840     if ((*cp == '\0') || (vp == NULL))
1841         return (0);                     /* disallow null string in cp */
1842     *vp = 0;
1843 again:
1844     /*
1845      * Collect number up to ``.''.  Values are specified as for C:
1846      *      0x=hex, 0=octal, other=decimal.
1847      */
1848     val = 0; base = 10;
1849     if (*cp == '0') {
1850         if (*++cp == 'x' || *cp == 'X')
1851             base = 16, cp++;
1852         else
1853             base = 8;
1854     }
1855     while ((c = *cp) != '\0') {
1856         if (isdigit(c)) {
1857             if ((c - '0') >= base)
1858                 break;
1859             val = (val * base) + (c - '0');
1860             cp++;
1861             continue;
1862         }
1863         if (base == 16 && isxdigit(c)) {
1864             val = (val << 4) + (c + 10 - (islower(c) ? 'a' : 'A'));
1865             cp++;
1866             continue;
1867         }
1868         break;
1869     }
1870     if (*cp == '.') {
1871         /*
1872          * Internet format:
1873          *      a.b.c.d
1874          *      a.b.c   (with c treated as 16-bits)
1875          *      a.b     (with b treated as 24 bits)
1876          */
1877         if ((pp >= parts + 3) || (val > 0xff)) {
1878             return (0);
```

```
1879         }
1880         *pp++ = (u_char)val;
1881         cp++;
1882         goto again;
1883     }
1884     /*
1885      * Check for trailing characters.
1886      */
1887     if (*cp != '\0' && !isspace(*cp)) {
1888         return (0);
1889     }
1890     /*
1891      * Concoct the address according to the number of parts specified.
1892      */
1893     n = pp - parts;
1894     switch (n) {
1895     case 0:                             /* a -- 32 bits */
1896         break;
1897     case 1:                             /* a.b -- 8.24 bits */
1898         if (val > 0xffffff)
1899             return (0);
1900         val |= parts[0] << 24;
1901         break;
1902     case 2:                             /* a.b.c -- 8.8.16 bits */
1903         if (val > 0xffff)
1904             return (0);
1905         val |= (parts[0] << 24) | (parts[1] << 16);
1906         break;
1907     case 3:                             /* a.b.c.d -- 8.8.8.8 bits */
1908         if (val > 0xff)
1909             return (0);
1910         val |= (parts[0] << 24) | (parts[1] << 16) | (parts[2] << 8);
1911         break;
1912     default:
1913         return (0);
1914     }
1915     *vp = val;
1916     return (cp - cp0);
1917 }

1919 /*
1920  * setxonxoff - modify the asyncmap to include escaping XON and XOFF
1921  * characters used for software flow control.  Returns 1 upon successful
1922  * processing of options, and 0 otherwise.
1923  */
1924 /*ARGSUSED*/
1925 static int
1926 setxonxoff(argv, opt)
1927     char **argv;
1928     option_t *opt;
1929 {
1930     int xonxoff = 0x000A0000;

1932     lcp_wantoptions[0].neg_asyncmap = 1;
1933     lcp_wantoptions[0].asyncmap |= xonxoff;     /* escape ^S and ^Q */
1934     lcp_allowoptions[0].asyncmap |= xonxoff;
1935     xmit_accm[0][0] |= xonxoff;
1936     xmit_accm[0][4] |= xonxoff;         /* escape 0x91 and 0x93 as well */

1938     crtscts = -2;
1939     return (1);
1940 }

1942 /*
1943  * setlogfile - open (or create) a file used for logging purposes.  Returns 1
1944  * upon success, and 0 otherwise.
```



```
1945  */
1946 /*ARGSUSED*/
1947 static int
1948 setlogfile(argv, opt)
1949     char **argv;
1950     option_t *opt;
1951 {
1952     int fd, err;

1954     if (!privileged_option)
1955         (void) seteuid(getuid());
1956     fd = open(*argv, O_WRONLY | O_APPEND | O_CREAT | O_EXCL, 0644);
1957     if (fd < 0 && errno == EEXIST)
1958         fd = open(*argv, O_WRONLY | O_APPEND);
1959     err = errno;
1960     if (!privileged_option)
1961         (void) seteuid(0);
1962     if (fd < 0) {
1963         errno = err;
1964         option_error("Can't open log file %s: %m", *argv);
1965         return (0);
1966     }
1967     if (log_to_file && log_to_fd >= 0)
1968         (void) close(log_to_fd);
1969     log_to_fd = fd;
1970     log_to_file = 1;
1971     early_log = 0;
1972     return (1);
1973 }

1975 #ifdef PLUGIN
1976 /*
1977  * loadplugin - load and initialize the plugin.  Returns 1 upon successful
1978  * processing of the plugin, and 0 otherwise.
1979  */
1980 /*ARGSUSED*/
1981 static int
1982 loadplugin(argv, opt)
1983     char **argv;
1984     option_t *opt;
1985 {
1986     char *arg = *argv;
1987     void *handle;
1988     const char *err;
1989     void (*init) __P((void));

1991     handle = dlopen(arg, RTLD_GLOBAL | RTLD_NOW);
1992     if (handle == NULL) {
1993         err = dlerror();
1994         if (err != NULL)
1995             option_error("%s", err);
1996         option_error("Couldn't load plugin %s", arg);
1997         return (0);
1998     }
1999     init = (void (*)(void))dlsym(handle, "plugin_init");
2000     if (init == NULL) {
2001         option_error("%s has no initialization entry point", arg);
2002         (void) dlclose(handle);
2003         return (0);
2004     }
2005     info("Plugin %s loaded.", arg);
2006     (*init)();
2007     return (1);
2008 }
2009 #endif /* PLUGIN */
```