Traceroute:
prologo di un attacco?

Autore:

Armando Leotta
armyz@idic.caos.it

Premesse

Non e' affatto raro sentirsi bersagliati dai mass-media che riportano in tono piuttosto allarmistico i nuovi attacchi a siti di multinazionali o inoperabilità di grosse catene di E-commerce ad opera di fantomatici gruppi chiamati spesso sommariamente hacker.
Cominciamo col fare una certa distinzione e chiarezza tra il significato di hacker e e di cracker, hacking ,exploit e Denial Of Service (DoS) e a quello che ci sta dietro a questi termini.

L'hacker e' un soggetto spesso esibizionista, tendenzialmente anarchico e che sfida i sistemi di sicurezza per il gusto di penetrarli senza alcuno scopo di lucro.
Un cracker invece non si limita certo ad inserire dei tag nelle homepage dei siti violati ma utilizza quel momentaneo accesso in qualsiasi modo possa ritornargli più conveniente.
Ad esempio, se un hacker si dovesse trovare di fronte ad un database di clienti paganti con carte di credito, molto probabilmente farebbe in modo da pubblicizzare negativamente quel servizio in ogni modo, lecito e non, mentre un cracker sfrutterebbe la situazione per fare acquisti gratuiti a discapito dei mal capitati clienti.

Anche tecnicamente, i tipi di "attacchi" menzionati precedentemente stabiliscono in maniera piuttosto chiara la capacita' dell'artefice nonché i mezzi a sua disposizione.

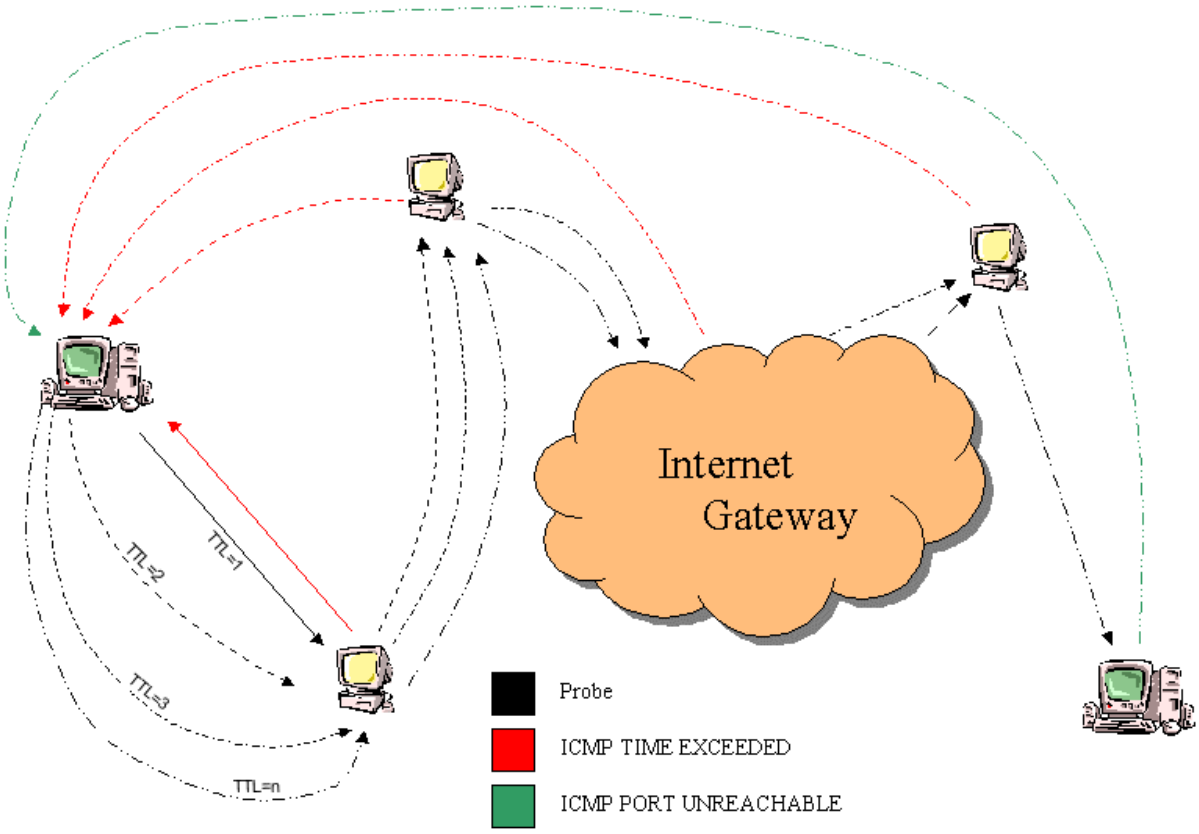
Un exploit e' un tipo di attacco che sfrutta bug conosciuti o meno di un determinato prodotto il che implica solitamente una discreta conoscenza del linguaggio C e del mondo Unix.
Diversamente, il Denial of Service e' un attacco alla portata praticamente di tutti essendo spesso causato da un sovraccarico di operazioni lecite. Classici esempi sono il ping flood, lo smurf, vari tipi di nuke, etc.
Tempo fa campeggiava su internet un' appuntamento online per floodare da ogni punto del mondo il sito ufficiale del vaticano, noti motori di ricerca, librerie, classici esempi di DoS per i quale non ci vuole una profonda conoscenza ne' di C ne' di unix bensì è sufficiente una connessione possibilmente veloce ed un programma di pochi kilobytes disponibile in ogni angolo remoto della rete.

Descrizione del problema

L'unica cosa che in effetti accomuna queste tipologie di attacchi cosi' profondamente diverse tra loro è la preparazione dell'attacco stesso: e' impensabile pensare di attaccare una macchina senza valutare il percorso necessario per raggiungerla.
Esiste una tecnica ormai consolidata e *standard de facto* chiamata **traceroute** che ha esattamente questo scopo:
ci permette di determinare il path che gli IP datagram seguono dal nostro host ad un altro.
Traceroute utilizza il campo TTL dell'IP e 2 ICMP message (vedi figura nella pagina seguente).
Inizialmente spedisce un UDP datagram all'host destinazione specificando un TTL = 1.
Il primo router, una volta ricevuto tale datagram e decrementato di un'unita' il TTL riducendolo cosi' a zero, effettua il cosiddetto *drop* del pacchetto restituendo al mittente un ICMP message di tipo TIME EXCEEDED congiuntamente all'indirizzo del router intermedio.

A questo punto viene spedito un altro UDP datagram questa volta con TTL incrementato di uno che, di fatto, localizza il successivo router nel path dal nostro host a quello destinazione.
Usando questa tecnica si costruisce hop by hop il percorso completo.

Quando il datagram raggiunge effettivamente la macchina destinazione quest'ultima restituirà un ICMP code di tipo PORT UNREACHABLE, dato che il *probe* viene spedito a porte random normalmente non utilizzate.
Esiste una variante che prevede l'utilizzo di un ICMP message di tipo ECHO REPLY al posto del datagramma UDP: in questo caso si elimina il rischio (comunque minimo) di trovare un servizio in ascolto sulla porta su cui si e'inviato il pacchetto.



Proposta

L'idea consiste nel tracciare tutti i traceroute entranti verso una macchina qualsiasi appartenente alla stessa subnet in cui risiede l'host che ospita il processo detector (ammesso che l'nterfaccia di rete supporti il modo promiscuo).

Implementazione

Il progetto e' realizzato sfruttando la possibilità di esaminare i pacchetti ricevuti dal datalink layer, combinata alla capacita' dell'interfaccia di rete di configurarsi nel cosiddetto promiscuous mode.
Tale capacita' permette all'applicazione di esaminare tutti i pacchetti transitanti nella sottorete e non solamente quelli destinati all' host nel quale il programma e' in esecuzione.

I tre metodi comuni per accedere al datalink layer su macchine Unix sono:
- il BSD Packet Filter (BPF),

- il SVR4 Data Link Provider Interface (DLPI)
- e i SOCK_PACKET interface di Linux.

Il progetto e' stato realizzato utilizzando la ***libpcap***[1], una libreria di pubblico dominio per la cattura dei pacchetti: essa funziona con tutte le implementazione appena descritte rendendo l'applicazione indipendente dall'effettivo datalink access fornito dal sistema operativo (cross-platform).

L'applicazione prevede la possibilità di effettuare il monitoring a più interfacce di rete attive sul sistema tramite la creazione di un thread (pthread) per ogni interfaccia.
Nel caso in cui un'interfaccia non sia attiva, disponibile o perfino esistente il thread incaricato del monitoring della stessa viene chiuso e la normale esecuzione continua per tutte le interfacce valide specificate in sede di avvio del programma (parametri a riga di comando).
Lo stesso risultato si raggiunge nel caso di un qualsiasi errore in qualsiasi momento da parte delle procedure di fetch delle libpcap.
Nel caso in cui non esistano più dei thread attivi (lo stesso avviene in caso di ricezione di SIGKILL / SIGTERM signal) il programma chiude *safely* eseguendo cioè tutte quelle procedure atte ad eliminare correttamente le strutture temporanee indispensabili per il multithreading (thread specific data e thread local storage) , la lista stessa contenente tutti i thread ancora aperti nonche' i MUTEX impiegati per la sincronizzazione dei thread stessi.

E' possibile specificare dei parametri a riga di comando come –d per forzare la daemonizzazione e –n, numeric only, per non risalire agli hostname degli ip tracciati oltre ai soliti –v per mostrare la versione di rtraced utilizzata e –h per avere l' help.
Tutto l'output generato viene diretto sulla syslog e, nel caso di interattività, viene inoltrato anche sullo stderr.

Il messaggio appare nella forma :
[**device**] [**type**] *traceroute attempt from* <**IP_sorgente**> (<**host_sorgente** | err |n/a)> *to* <**IP_dest**> (**host_dest** | err | n/a) *at* <**localtime**><**timezone**>
dove :

- **device** e' l'interfaccia sulla quale e' pervenuta la richiesta;
- **type** e' il tipo di traceroute utilizzato: UDP (unix) o ICMP (windows e alcuni *BSD);
- **IP_sorgente** e' l'ip che ha effettuato la richiesta di traceroute;
- **Host_sorgente** e' l'hostname associato all'IP che ha effettuato la richiesta di traceroute (err o n/a in caso di mancata risoluzione;
- **IP_dest** e' l'ip del tracciato: non sempre e' l'IP della macchina su cui il programma è eseguito in quanto si e' in grado di rilevare delle richieste volte a tutte le workstation appartenenti alla stessa subnet alla quale appartiene la macchina su cui il processo detector è in esecuzione;
- **Host_dest** e' l'hostname associato a IP_dest;
- **Localtime** e **timezone** identificano chiaramente l'ora e la timezone al momento del traceroute, indispensabile in caso di eventuale contestazione e/o chiarimenti. La timezone e' di estrema necessità nel caso di richiesta spiegazioni verso admin esteri (e' buona norma documentare al massimo, full Timestamping).

Si sono risolte alcune problematiche (personalmente) non preventivate:

- Incongruenza del datalink type nelle libpcap:
  si è giunti alla conclusione che esiste un errore nelle #defines riguardante il datalink relativo all'interfaccia ppp (le define associano un intero diverso a quello restituito in realtà).
  Ho personalmente contattato gli sviluppatori segnalando il bug;
- "fake" traceroute spediti da parte del default gateway in caso di connessione ppp :
  si è codificata una funzione apposita che, passando in rassegna tutte le interfacce attive del sistema, nel caso di connessione Point-to-Point restituisce il default gateway .
  Se ,in seguito ad un apposito confronto, la richiesta in questione risulta pervenire dal default gateway si evitano ulteriori controlli in quanto non si tratta di una *vera* richiesta di trace.

[1] La libreria è pubblicamente disponibile presso ftp://ftp.ee.lbl.gov/libpcap.tar.Z

Questa situazione si crea perche' il gateway confeziona delle richieste ICMP per assicurarsi che il client sia ancora attivo (alive) ma lo fa inizializzando il valore del TTL = 1, in quanto sicuro che, essendo il gateway per quell'IP, non può essere più "lontano": da qui la necessità di *filtrare* tali richieste.

Conclusioni

Non sarà certo il programma che farà diminuire il numero di attacchi ma, avere qualche log in più, può solamente essere di beneficio, magari punto di partenza per ricerche, spesso a posteriori, per risalire agli autori dell'attacco stesso oppure se integrato da suite di sicurezza dinamiche puo' costituire un interessante strumento di alert.
Il risultato sembra essere un programma abbastanza leggero, robusto, utile e di facile utilizzo.
Inoltre, l'implementazione dei pthread lo rende ancora più versatile in quanto è possibile effettuare il monitoring di n interfacce da un unico processo.
Si include alla presente il sorgente rtraced.c.

Source

```
/*
 * rtraced
 * copyright (c) 2000 Armando 'ArMyZ' Leotta. all rights reserved.
 *
 * web
 *          http://armyz.idic.caos.it/rtraced/
 * author:
 *          Armando 'ArMyZ' Leotta
 *
 * author email:
 *          armyz@idic.caos.it
 *
 * description:
 *          Reveal if the host is currently traced and the tracing ip host.
 *          In promiscuous mode it reveals each trace attempt for all workstation of your subnet.
 *
 * compiling flags:
 *      none
 *
 * compiler suggested:
 *          GNU C/C++ Compiler 2.91+
 *
 * linker options
 *          libpcap pthread
 *
 * known working platforms:
 *          Linux Kernel >2.2.x
 *
 * version:
 *          rtraced v1.0.2(multithreaded)
 *
 * main changes
 *          added multithread support, fixed some minor bugs, improved readability and indented.
 *
 * history:
 *          08 Jul 2000         - added thread local storage and thread specific function
 *                                to handle reentrant data.
 *                              - moved all global variables into their respectively function
 *                                to be thread-aware.
 *                              - implementation and use of mutex locks to handle thread local
 *                                storage list.
 *                              - used snprintf in place of sprintf to avoid buffer overflows.
 *                              - handling of the SIGKILL and SIGTERM signals.
 *                              - added an infinite loop, with latency using usleep to cycle
 *                                threads, in main function.
 *          09 Jul 2000         - added multithread support to process_res and process_ip.
 *                              - used return in place of exit in main function (so ansi suggests).
 *                              - added attributes for the creating thread (detached and RR
 *                                (Round Robin) scheduling, stack size).
 *                              - fixed the case no threads are created and removed the use
 *                                of the first arg as device to do the main loop.
 *                              - outputs now is on stderr, fprintf used all around.
 *                              - fixed some minor bugs (order of headers, changed my_err and
 *                                my_log to handle function name, using of fixed value to store
 *                                error value in the host_res field, the name of the application
 *                                printed out now is taken from MY_LOG_APP define).
 *                              - created MUTEX_XXX macros.
 *                              - added INVALID_HOST_RES definition.
 *                              - added SLEEP_LATENCY for the sleep latency in main loop.
```

```
 *                      - added the reassignment of the default behaviour to the signal
 *                        SIGKILL for the created threads, to avoid parent process,
 *                        running in daemon mode, get killed.
 *      10 Jul 2000     - added the feature to ignore ( closing ) a threads running for
 *                        an invalid interface; if for any reason the pcap process failed
 *                        we'd see the single thread serving it closed ( and others keep
 *                        running).
 *                      - fixed the case no more working threads are running for us.
 *                      - source indented and commented as much as possible the new
 *                        function, restyled the comments in C style.
 *                      - fixed signal handler for foreground mode.
 *                      - added Timestamp feature.
 */

#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/if_ether.h>
#include <net/if.h>
#include <net/bpf.h>
#include <pcap.h>
#include <syslog.h>
#include <pthread.h>

#define ETHHDR_SIZE             14
#define PPPHDR_SIZE             4
#define SLIPHDR_SIZE            16
#define RAWHDR_SIZE             0
#define LOOPHDR_SIZE            4
#define FDDIHDR_SIZE            21
#define DEVNULL                         "/dev/null"
#define MY_LOG_APP              "rtraced"
#define  MY_LOG_LOG             LOG_AUTHPRIV
#define  MY_LOG_LEV             LOG_INFO
#define PPPX                    "ppp"
#define TRACE_VER               "ver. 1.0.2(multithreaded) - Armando 'ArMyZ' Leotta - armyz@idic.caos.it"
#define MAX_BUFF                1024
#define SNAPLEN                         4096
#define PROMISC                         1
#define READ_TIMEOUT                    1000
#define OPTIMIZE                1
#define LOOP                    -1
#define STACK_SIZE              65536
#define SLEEP_LATENCY                   10
#define DEV_SIZE                32

#define INVALID_HOST_RES        ((char*)0xdeadbeef)
```

```
#define MUTEX_DESTROY(lock)     pthread_mutex_destroy(&(lock));
#define MUTEX_ACQUIRE(lock)     pthread_mutex_lock(&(lock));
#define MUTEX_RELEASE(lock)     pthread_mutex_unlock(&(lock));


/* Simple linked list to store thread-specific informations */
typedef struct _thread_tls {
        pthread_t       thread_id;              /* Thread ID */
        char            device[DEV_SIZE];       /* 32 should be enough */
        char*           host_res;               /* Host Result */
        char*           src;                    /* Source IP in x.x.x.x format */
        struct _thread_tls* next;               /* Next entry */
} thread_tls;

static int daem = 0;
static int resolve = 1;
static volatile int  quit = 0; /* Volatile, because it will be changed through the multithreading
                                * environment.
                                */
static int global_error_return = 0;
/* Mutex to access tls variable */
static pthread_mutex_t tls_lock = PTHREAD_MUTEX_INITIALIZER;
/* Global to this module tls list variable */
static thread_tls* tls_list = NULL;
static int threads_closed = 0;
/* Gloabal to keep count of the threads actually running

/* prototypes */
static void my_err(char* func, char *err);
static void my_log(char* func, char *my_str);
static unsigned int def_gw(void);
static char* process_res(unsigned int ip_traced);
static void process_ip(const struct ip * ip, int len);
static thread_tls* tls_create(pthread_t tid, char* device);
static thread_tls* tls_get_current(void);
static void tls_kill_threads(void);
static void tls_destroy(void);
static void signal_quit(int sig);
static void daemonize(void);
static void loopback_interface(u_char * info, struct pcap_pkthdr * header, u_char * data);
static void eth_interface(u_char * info, struct pcap_pkthdr * header, u_char * data);
static void ppp_interface(u_char * info, struct pcap_pkthdr * header, u_char * data);
static void my_pcap_thread_function(char *device);
static void version(void);
static void usage(int e);
static void* thread_init(void* device);



/*
 * prototype:
 *      static void my_err(char* func, char* err)
 *
 * description:
 *      handles and sends on the my_log function the error
 *
 * outputs:
 *      none
 *
 * inputs:
 *      char* func      - the function caller [optional but useful for debugging]
```

```
 *          char* err           - the error string
 */
static void
my_err(char* func, char *err)
{
          char buf[MAX_BUFF];    /* Local copy buffer */

      MUTEX_ACQUIRE(tls_lock);      /* Don't let try another thread interfere with us*/
          if(func!=NULL) snprintf(buf, sizeof(buf), "<%s::%s> %s", MY_LOG_APP, func, err);
          else snprintf(buf, sizeof(buf), "<%s> %s", MY_LOG_APP, err);
          my_log(NULL, buf);


      MUTEX_RELEASE(tls_lock);      /* We can release it now */
}

/*
 * prototype:
 *          static void my_log(char* func, char* my_str)
 *
 * description:
 *          log into the syslog the message
 *
 * outputs:
 *          none
 *
 * inputs:
 *          char* func          - the function calling [optional but useful for debugging]
 *          char* my_str        - the error string to log into the syslog
 */
static void
my_log(char* func, char *my_str)
{
          char buf[MAX_BUFF];    /* Local copy buffer */

          if(func!=NULL) snprintf(buf, sizeof(buf), "<%s::%s> %s", MY_LOG_APP, func, my_str);
          else snprintf(buf, sizeof(buf), "%s", my_str);

          syslog(MY_LOG_LEV, my_str);
          /* If not running in daemon mode or (beeing not in daemon mode)
                    current host_res is invalid, print it out on the stderr */

          if ((!daem) || (!daem && tls_get_current()->host_res != INVALID_HOST_RES))
                    fprintf(stderr, "%s\n", buf);

}

/*
 * prototype:
 *          static thread_tls* tls_create(pthread_t tid, char* device)
 *
 * description:
 *          allocs, sets up and chains a new tls for thread-specific information
 *          containing thread-specific host_res and thread-specific device name.
 *
 * outputs:
 *          the tls information descriptor pointer
 *
 * input:
 *          pthread_t tid       - the thread id (can be gained with pthread_self, but for the sake of correctness, we pass it
forcely)
 *          char* device        - the device name
```

```
 */
static thread_tls*
tls_create(pthread_t tid, char* device)
{
	thread_tls* aux;

	MUTEX_ACQUIRE(tls_lock);	/* First of all, don't let try another thread interfere with us */

	/* Allocate a tls entry */
	if((aux = (thread_tls*)malloc(sizeof(thread_tls)))==NULL) {
		my_err("tls_create", "Memory exhausted");
	}
	/* Fill it with zeroes */
	memset(aux, 0, sizeof(thread_tls));
	/* Chain it and set it up */
	aux->next = tls_list;
	strncpy(aux->device, device, sizeof(aux->device));	/* Don't let try to overrun the buffer! */
	aux->host_res = INVALID_HOST_RES;
	aux->src = NULL;
	aux->thread_id = tid;					/* Get the thread id */
	tls_list = aux;

	MUTEX_RELEASE(tls_lock);		/* Release the mutex */

	return aux;
}

/*
 * prototype:
 *	static thread_tls* tls_get_current(void)
 *
 * description:
 *	get the tls for the current thread, this function has to be inside a thread!
 *
 * outputs:
 *	the tls information descriptor pointer containing the current thread informations.
 *
 * input:
 *	none
 */
static thread_tls*
tls_get_current(void)
{
	thread_tls* list;

	if(pthread_self()<=0) return NULL;		/* If we aren't in a thread, just return a NULL pointer */

	MUTEX_ACQUIRE(tls_lock);			/* Lock the mutex : we need synchronization with tls_list
global var! */

	list = tls_list;
	while(list) {
		if(list->thread_id == pthread_self()) {	/* Is the entry the current thread? */
			MUTEX_RELEASE(tls_lock);	/* We have done */
			return list;
		}
		list = list->next;			/* Move to the next entry */
	}

	MUTEX_RELEASE(tls_lock);			/* We have done */

	return NULL;					/* Shouldn't be possible get here, anyway handle it! */
```

```
}


/*
 * prototype:
 *          static void tls_kill_threads(void)
 *
 * description:
 *          kills all threads in the tls list
 *
 * outputs:
 *          none
 *
 * input:
 *          none
 */
static void
tls_kill_threads(void)
{
          thread_tls* list;

          MUTEX_ACQUIRE(tls_lock);      /* Expecially while killing threads, it's safe to shutdown all the request to
the mutex */

          list = tls_list;
          while(list) {
                    if(list->thread_id>0) {
                              if(list->thread_id != pthread_self()) { /* If the entry is not the current thread */
                                        void* result;

                                        pthread_kill(list->thread_id, SIGKILL);       /* Raise a SIGKILL signal to the thread
*/
                                        pthread_join(list->thread_id, &result);       /* Join it and wait for its dead! */
                                        list->thread_id = 0;
                              }
                    }
                    list = list->next;
          }

          MUTEX_RELEASE(tls_lock);
}

/*
 * prototype:
 *          static void tls_destroy(void)
 *
 * description:
 *          kills all threads and frees up the linked list
 *
 * outputs:
 *          none
 *
 * input:
 *          none
 */
static void
tls_destroy(void)
{
          thread_tls* list, * next;
          pthread_t calling_tid = 0;
          char* src;
```

```
	tls_kill_threads();				/* First kill all threads */

	MUTEX_ACQUIRE(tls_lock);		/* Lock the mutex, we're about to destroy everything!
						 * Side note: Locking mutex in this phase has no effect,
						 * it has been done just to be sure someone wants to access
						 * to the tls_list from a tls-specific function...
						 */
	list = tls_list;
	while(list) {
		next = list->next;		/* Get the next entry */
		src = list->src;		/* We would scratch the content of list freeing directly list->src */
		if(src) free(src);		/* Frees up the strdup */
		free(list);			/* Free it up! */
		list = next;			/* Move to the next saved */
	}
	tls_list = NULL;			/* tls_list is now invalidated! */

	MUTEX_RELEASE(tls_lock);		/* Release the mutex, now that tls_list is invalidated, further access
						 * to it will result in NULL pointer :)
						 */
}

/*
 * prototype:
 *	static void signal_quit(int sig)
 *
 * description:
 *	handler for SIGTERM and SIGKILL signals
 *
 * outputs:
 *	none
 *
 * input:
 *	sig		- The signal number
 */
static void
signal_quit(int sig)
{


	quit = 1;				/* Set's global variable quit to 1, so the main loop will stop */
}


/*
 * prototype:
 *	static void daemonize(void)
 *
 * description:
 *	switch the application into background forking
 *
 * outputs:
 *	none
 *
 * input:
 *	none
 */
static void
daemonize(void)
{
	int		fd;
	pid_t		pid;
```

```
	struct sigaction sa;


	if ((pid = fork()) != 0) {
		if (pid == -1)
			my_err("daemonize", "Unable to daemonize.");
		exit(0);
	}
	/* parents terminates & 1st child continues */


	/*
	 * Create a new session.
	 */
	if (setsid() == -1)
		my_err("daemonize", "Unable to set session leader.");

	/*
	 * Sets up signal SIGHUP handler
	 */
	sa.sa_handler = SIG_IGN;
	sa.sa_flags = 0;
	sigaction(SIGHUP, &sa, NULL);
	/*
	 * Sets up signal SIGKILL and SIGTERM handler
	 */
	sa.sa_handler = signal_quit;
	sa.sa_flags = 0;
	sigaction(SIGKILL, &sa, NULL);
	sigaction(SIGTERM, &sa, NULL);


	if ((pid = fork()) != 0)	/* 1st child terminates & 2nd child continues */
		exit(0);
	if (chdir("/tmp") == -1)
		my_err("daemonize", "Unable to chroot directory");
	umask(0);
	if ((fd = open(DEVNULL, O_RDWR, 0)) != -1) {
		dup2(fd, STDIN_FILENO);
		dup2(fd, STDOUT_FILENO);
		dup2(fd, STDERR_FILENO);
		if (fd > 2)
			close(fd);
	}

}


/*
 * prototype:
 *	static unsigned int def_gw(void)
 *
 * description:
 *	gets the PPP-device default gateway
 *
 * outputs:
 *	the default gateway ip in unsigned integer value
 *
 * input:
 *	none
 */
static unsigned int
```

```
def_gw(void)
{
	int		sock_gw, i, if_count;
	char		buff[MAX_BUFF];
	struct sockaddr_in sin;
	struct ifreq	*ifr;
	struct ifconf	ifc;

	if ((sock_gw = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
		perror("<traced::def_gw> Socket creation: ");
		exit(1);
	}
	ifc.ifc_len = sizeof(buff);
	ifc.ifc_buf = buff;
	if (ioctl(sock_gw, SIOCGIFCONF, &ifc) < 0) {
		perror("ioctl error");
		exit(1);
	}
	ifr = ifc.ifc_req;
	for (i = ifc.ifc_len / sizeof(struct ifreq); --i >= 0; ifr++) {

		if (strncmp(ifr->ifr_name, PPPX, sizeof(PPPX)) != 0)
			continue;

		if (ioctl(sock_gw, SIOCGIFFLAGS, ifr) < 0) {
			perror("ioctl error");
			break;
		}
		if (ifr->ifr_flags & IFF_LOOPBACK)
			continue;		/* Ignore loopbacks */

		/*
		 * snprintf(temp, sizeof(temp), "P-t-P Interface probed and found: %s\n",
		 * ifr->ifr_name);
		 */
		/* my_log(temp); */
		if (ifr->ifr_flags & IFF_POINTOPOINT) {
			if (ioctl(sock_gw, SIOCGIFDSTADDR, ifr) < 0) {
				perror("ioctl error");
				break;
			}
			/* fprintf(stderr, "%s\n", ifr->ifr_name); */
			sin = *(struct sockaddr_in *) & ifr->ifr_dstaddr;
		}
		return sin.sin_addr.s_addr;
	}
	return 0;
}


/*
 * prototype:
 *	static char* process_res(unsigned int ip_traced)
 *
 * description:
 *	resolve the name of the traced ip or "err" if failed or
 *	"n/a" if it can't be resolved.
 *
 * outputs:
 *	the resolved name
 *
 * input:
```

```
 *          ip_traced           - the ip in unsigned integer value
 */
static char         *
process_res(unsigned int ip_traced)
{
        struct hostent *h_resol;
        char   temp[MAX_BUFF];
        thread_tls* tls;

        tls = tls_get_current();                /* Get current thread tls */
        if ((h_resol = gethostbyaddr((char const *) &ip_traced, sizeof(struct in_addr), AF_INET)) == NULL) {
                tls->host_res = INVALID_HOST_RES;
                snprintf(temp, sizeof(temp), "gethostname error for host: %s %s", tls->src, hstrerror(h_errno));
                my_log("process_res", temp);
        } else if (h_resol->h_name != NULL)
                tls->host_res = h_resol->h_name;

        else
                tls->host_res = "n/a";

        return (tls->host_res==INVALID_HOST_RES)?"err":tls->host_res;
}

/*
 * prototype:
 *          static void process_ip(const struct ip * ip, int len)
 *
 * description:
 *          process an ip packet
 *
 * outputs:
 *          none
 *
 * input:
 *          ip                  - ip packet header
 *          len                 - length of the packet
 */
static void
process_ip(const struct ip * ip, int len)
{
        struct icmp    *icmp;
        struct hostent *h_res;
        char           buf[MAX_BUFF];
        char          *buf2 = buf;
        char          *tmp, *trace_p;
        char                  temp[MAX_BUFF];
        thread_tls*           tls;
      time_t ticks;
        int           hlen1;

        tls = tls_get_current();                /* Get current thread tls */

        if (ip->ip_ttl == 1) {

                if (ip->ip_src.s_addr == def_gw()) {
                        return;
                }
                switch (ip->ip_p) {         /* netinet/ip_icmp.h */

                case IPPROTO_UDP:
                        {
                                /* (17) */
```

```
                                trace_p = "UDP";
                                break;
                        }

                case IPPROTO_ICMP:{
                                /* (1) */
                                icmp = (struct icmp *) ((u_char *) ip + ip->ip_hl * 4);
                                if (icmp->icmp_type != ICMP_ECHO)
                                        return;
                                /* ICMP based traceroute uses ICMP_ECHO(8) */
                                trace_p = "ICMP";
                                break;
                        }

                default:
                        return;
                }

                tls->src = strdup(inet_ntoa(ip->ip_src));
                ticks = time(NULL);

                if (!resolve) {
                        snprintf(temp, sizeof(temp), "\n[%s] [%s] traceroute attempt from %s to %s at %.24s
%s\r\n\n", tls->device, trace_p, tls->src, inet_ntoa(ip->ip_dst),ctime(&ticks),*tzname);
                        my_log(NULL, temp);
                } else {
                        /* i hate no REEntrant functions */
                        buf2 += sprintf(buf2, "\n[%s] [%s] traceroute attempt from %s (%s)", tls->device, trace_p,
tls->src, process_res(ip->ip_src.s_addr));
                        buf2 += sprintf(buf2, " to %s (%s) at %.24s %s\r\n\n", inet_ntoa(ip->ip_dst), process_res(ip-
>ip_dst.s_addr),ctime(&ticks),*tzname);
                        my_log(NULL, buf);
                }
        }
}


/*
 * prototype:
 *      static void loopback_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
 *
 * description:
 *      handler for loopback interface packets      (see pcap.h)
 *
 * outputs:
 *      none
 *
 * input:
 *      info            - not actually used
 *      header          - packet header
 *      data            - raw packet data
 */
static void
loopback_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
{


        process_ip((const struct ip *) (data + LOOPHDR_SIZE),      /* len is the length of the packet(off wire) see
pcap.h */
                   (int) (header->len - LOOPHDR_SIZE));
}
```

```
/*
 * prototype:
 *          static void eth_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
 *
 * description:
 *          handler for ethernet interface packets          (see pcap.h)
 *
 * outputs:
 *          none
 *
 * input:
 *          info                - not actually used
 *          header              - packet header
 *          data                - raw packet data
 */
static void
eth_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
{
          struct ether_header *eth_p = (struct ether_header *) data;



          if (ntohs(eth_p->ether_type) == ETHERTYPE_IP)   /* see net/ethernet.h */
                    process_ip((struct ip *) (data + ETHHDR_SIZE), (int)
                              (header->len - ETHHDR_SIZE));

}

/*
 * prototype:
 *          static void ppp_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
 *
 * description:
 *          handler for ppp interface packets    (see pcap.h)
 *
 * outputs:
 *          none
 *
 * input:
 *          info                - not actually used
 *          header              - packet header
 *          data                - raw packet data
 */
static void
ppp_interface(u_char * info, struct pcap_pkthdr * header, u_char * data)
{


          process_ip((struct ip *) (data + RAWHDR_SIZE), (int)
                    (header->len - RAWHDR_SIZE));
}

/*
 * prototype:
 *          static void my_pcap_thread_function(char *device)
 *
 * description:
 *          device processing thread
 *
 * outputs:
 *          none
 *
```

```
 * input:
 *      device          - the device name
 */
static void
my_pcap_thread_function(char *device)
{
        /* XXX: Take a look at pcap manual online  - 256 (in pcap.h) */
        struct bpf_program fp;
        pcap_handler    p_handler;
        struct in_addr  net, mask;
        pcap_t      *pktd;
        char            buff[MAX_BUFF];
        char            temp[MAX_BUFF], *str = NULL;
        char            err_buff[PCAP_ERRBUF_SIZE];
        u_char      *pkt_data = NULL;


        if (!(pktd = pcap_open_live(device, SNAPLEN, PROMISC, READ_TIMEOUT, err_buff))) {

                my_err("my_pcap_thread_function open_live", err_buff);
                snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing thread.\n",device);
                my_log(NULL,temp);
                MUTEX_ACQUIRE(tls_lock);
                threads_closed++;
                MUTEX_RELEASE(tls_lock);
                pthread_exit(NULL);

         }
        if (pcap_lookupnet(device, &net.s_addr, &mask.s_addr, err_buff) == -1){
                my_err("my_pcap_thread_function lookupnet", err_buff);
                snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing thread.\n",device);
                my_log(NULL,temp);
                MUTEX_ACQUIRE(tls_lock);
                threads_closed++;
                MUTEX_RELEASE(tls_lock);
                pthread_exit(NULL);
         }

        if (!daem) {                            /* It could mess up the stderr output*/
                fprintf(stderr, "interface: %s", device);
                if (net.s_addr && mask.s_addr) {
                        fprintf(stderr, " (%s/", inet_ntoa(net));
                        fprintf(stderr, "%s)", inet_ntoa(mask));
                }
                fprintf(stderr, "\n\n");
        }


        if (pcap_compile(pktd, &fp, str, OPTIMIZE, mask.s_addr) < 0) {
                my_err("my_pcap_thread_function compile", pcap_geterr(pktd));
                snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing thread.\n",device);
                my_log(NULL,temp);
                MUTEX_ACQUIRE(tls_lock);
                threads_closed++;
                MUTEX_RELEASE(tls_lock);
                pthread_exit(NULL);
                }
        /*
         * fp is a pointer to an array of bpf_program structure ,here the
         * pcap_compile call result
         */
```

```
	if (pcap_setfilter(pktd, &fp) == -1) {
		my_err("my_pcap_thread_function setfilter", pcap_geterr(pktd));
		snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing thread.\n",device);
		my_log(NULL,temp);
		MUTEX_ACQUIRE(tls_lock);
		threads_closed++;
		MUTEX_RELEASE(tls_lock);
		pthread_exit(NULL);
		}

	switch (pcap_datalink(pktd)) {	/* define in bpf.h */

	case DLT_NULL:
		{

			p_handler = (pcap_handler) loopback_interface;	/* NULL datalink, no
									 * link - layer
									 * encapsulation->lookpba
									 * ck interface */
			break;

		}

	case DLT_EN10MB:
		{
			p_handler = (pcap_handler) eth_interface;
			break;
		}

	case DLT_RAW:
		{
			/*
			 * case 12:		{ PPP datalink but handled as raw
			 * IP:probably an incorrect define in
			 * bpf.h(developers contacted)
			 */
			p_handler = (pcap_handler) ppp_interface;
			break;
		}
	default:{
			snprintf(temp, sizeof(temp), "Fatal error: unknown or unsupported data link");
			my_log("my_pcap_thread_function", temp);
			if (!daem)
				my_err("my_pcap_thread_function", temp);
			snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing
thread.\n",device);
			my_log(NULL,temp);
			MUTEX_ACQUIRE(tls_lock);
			threads_closed++;
			MUTEX_RELEASE(tls_lock);
			pthread_exit(NULL);
		}
	}

	(void)tls_create(pthread_self(), device);		/* Chain a new tls based on local info */

	usleep(SLEEP_LATENCY);
	if (pcap_loop(pktd, LOOP, p_handler, pkt_data) == -1) {
		/*
		 * A negative value(2 nd parameter) causes pcap_loop to loop
		 * forever(or until an error occurs)
		 */
```

```
                        my_err("my_pcap_thread_function", pcap_geterr(pktd));
                        snprintf(temp, sizeof(temp),"\nIgnoring request for invalid device *%s*,closing thread\n",device);
                        my_log(NULL,temp);
                        MUTEX_ACQUIRE(tls_lock);
                        threads_closed++;
                        MUTEX_RELEASE(tls_lock);
                        pthread_exit(NULL);
                        }

                /* closing descriptor */

                pcap_close(pktd);

}


/*
 * prototype:
 *          static void version(void)
 *
 * description:
 *          prints out the version and exit
 *
 * outputs:
 *          none
 *
 * input:
 *          none
 */
static void
version(void)
{
        fprintf(stderr, "\n%s: %s\n\n", MY_LOG_APP, TRACE_VER);
        exit(0);
}

/*
 * prototype:
 *          static void usage(int e)
 *
 * description:
 *          prints out the usage and exit program
 *
 * outputs:
 *          none
 *
 * input:
 *          e                   - error code to exit back to the shell
 */
static void
usage(int e)
{
        fprintf(stderr, "\nUsage: %s [-dvnh] <interface> [<interface> [...]]\n", MY_LOG_APP);
        fprintf(stderr, "-d:\tRun as (d)aemon;\n");
        fprintf(stderr, "-v:\tPrint (v)ersion only;\n");
        fprintf(stderr, "-n:\tDo (n)ot resolve detected ip's tracerouters;\n");
        fprintf(stderr, "-h:\tPrint (this) help;\n\n");
        exit(e);
}


/*
```

```
 * prototype:
 *          static void* thread_init(void* device)
 *
 * description:
 *          the thread function
 *
 * outputs:
 *          NULL pointer, should never return this function
 *
 * input:
 *          device              - the device name pointer
 */
static void    *
thread_init(void* device)
{
          struct sigaction sa;



          /*
           * Fix SIGKILL for the thread, if we don't do this,
           * we would kill the parent process if in daemon mode,
           * because would be called function signal_quit()
           * during the execution of function tls_kill_threads().
           * Side note: This won't change the default behaviour if
           * we are running foreground.
           */
          sa.sa_handler = SIG_DFL;
          sa.sa_flags = 0;
          sigaction(SIGKILL, &sa, NULL);



          my_pcap_thread_function((char*)device);
          return NULL;      /* We should never get here */

}

/*
 * prototype:
 *          int main(int argc, char**argv)
 *
 * description:
 *          main routine
 *
 * outputs:
 *          exiting error code
 *
 * input:
 *          argc                - arguments count
 *          argv                - arguments vector
 */
int
main(int argc, char **argv)
{
          int           c, i, index, failed,arg1;
          pthread_t       tid;
          pthread_attr_t      t_attr;               /* Threads attributes */
          char          *argloop = NULL;
          char                  temp[MAX_BUFF];
          struct sigaction sa;
```

```
	arg1 = 0;
	while ((c = getopt(argc, argv, "dnvh")) != EOF) {
		switch (c) {
		case 'd':
			daem = 1;
			break;
		case 'n':
			resolve = 0;
			break;
		case 'v':
			version();
			break;
		case 'h':
			usage(0);
			break;
		default:
			usage(-1);
			break;
		}
	}				/* while getopt */
	if (argc < 2) {
		usage(0);
		return -1;
	}

	openlog(MY_LOG_APP, LOG_NDELAY, MY_LOG_LOG);
	if (daem) {
		snprintf(temp, sizeof(temp), "\n%s is monitoring your specified interface(s).\n\nYou ran it as daemon:
don't specify -d flag to run it interactively.\n", MY_LOG_APP);
		my_log(NULL, temp);
		fprintf(stderr, "%s", temp);
		fprintf(stderr, "Daemonizing...\n");
		daemonize();
		snprintf(temp, sizeof(temp), "[%s] monitoring as daemon.\n", MY_LOG_APP);
		my_log(NULL, temp);
	} else {
		snprintf(temp, sizeof(temp),"\n[%d] %s is monitoring your specified interface(s).\n\nYou ran it
interactively, specify -d flag to run it as daemon.\n", getpid(),MY_LOG_APP);
		my_log(NULL, temp);

	}

	sa.sa_handler = signal_quit;
	sa.sa_flags = 0;
	sigaction(SIGKILL, &sa, NULL);
	sigaction(SIGTERM, &sa, NULL);

	index = optind;
	failed = 0;
	pthread_attr_init(&t_attr);			/* Initialize the thread attributes */
	pthread_attr_setstacksize(&t_attr, STACK_SIZE);	/* Set thread stack size to 64K */
	pthread_attr_setschedpolicy(&t_attr, SCHED_RR);	/* Set round robin scheduling policy */
	pthread_attr_setdetachstate(&t_attr, PTHREAD_CREATE_DETACHED); /* Create the thread already
detached */
	while (argv[index]) {
		argloop = argv[index++];
		arg1++;
		if (pthread_create(&tid, &t_attr, thread_init, (void*)argloop) != 0) {
		perror("thread_create");
		failed++;
	  }
```

```
	}

	pthread_attr_destroy(&t_attr);			/* Destroy the thread attributes */

	if(failed==(index-optind)) { /* All threads failed? */
		my_err(NULL, "All threads failed to create!");
		signal_quit(1);
	}


	while(!quit) {
		usleep(SLEEP_LATENCY);		/* Main loop --with latency-- */
		if (arg1 - threads_closed - failed == 0) signal_quit(1);
    }

	tls_destroy();	/* If quit is 1 (due to signal SIGKILL or SIGTERM), destroy tls_list and the mutex */
	MUTEX_DESTROY(tls_lock);

	/* We're closing, let's communicate it to the world */
	snprintf(temp, sizeof(temp), "%s exiting with rcode=%d.\n", MY_LOG_APP, global_error_return);

	if(!daem) fprintf(stderr, "%s", temp);
	my_log(NULL, temp);
	closelog();		/* closing syslog descriptor(optional but recommended) */

	return global_error_return;		/* return global_error_return */
}
```