Raw Sockets sous Windows

– Tolwin –

Introduction

Je vais essayer de faire un texte assez complet sur les raw sockets, un de ceux que j'aurais peut-être aimé à trouver lorsque je ne connaissais pas le domaine et que j'écumais le web à la recherche de ressources. En général la moisson est anglophone et linuxienne, ce qui nécessite des efforts de lecture et d'adaptation. Ca ne veut pas dire que ce texte, francophone et windowsien, se lira aussi facilement que Harry Potter !

Au cours de mes explications, je vais suivre un plan qui va du plus simple au plus compliqué, en me basant sur l'exemple d'un programme de trace de route. Chaque chapitre sera divisé en trois parties. Pour commencer, présentation du protocole et détails de l'implémentation. Ensuite, utilité de la manipulation du protocole. Et enfin, un un code source complet et fonctionnel. Les exemples ont été dévelopés et testés chez moi avec un Visual C++6 avec service pack donc c'est sûr, ca marche.

1. Examen de Tracert.exe :
Je vais commencer par examiner tracert.exe qui est fourni par Microsoft. Après avoir regardé ses entrailles, j'essaierais de me faire une idée de son fonctionnement.

2. Premières notions sur les raw sockets (ICMP).
Dans ce chapitre, je ne manipulerais que le protocole ICMP. Simple à comprendre, ce n'est donc pas sa difficulté qui se mettra en travers de la compréhension des raw sockets en eux-mêmes.

3. Les raws sockets dans le protocole IP :
Après avoir regardé ce qui est le protocole fondateur d'Internet, vous serez convaincus que contrairement aux rumeurs, Windows permet bien au programmeur de manipuler l'entête IP. Vous saurez également exactement de quoi il en retourne lorsqu'on parle d'IP Spoofing.

4. Execution du traceur de Route :
Pour finir, je ferais fonctionner le traceur de route en utilisant le protocole TCP. Vous aurez ainsi une bonne vue des protocoles les plus courants sur Internet, et les autres comme UDP vous seront très faciles à assimiler.

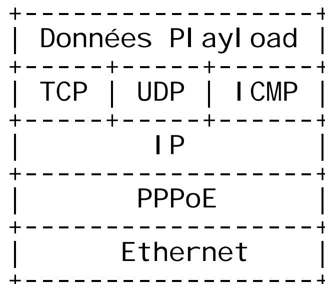
Je donnerais un résumé suffisant pour en comprendre la logique et le fonctionnement des protocoles mais je ne rentrerais pas dans les détails et les subtilités. Chaque protocole est expliqué en long, en large et en travers dans les RCF. Je vous invite à consulter http://abcdrfc.free.fr qui contient les RFC en français.

Pour bien profiter de ce texte vous aurez besoin :

- d'une cervelle en bon état
- de temps devant vous
- de Windows 2000 minimum
- de visual C++
- de pratiquer, d'essayer. Lire ne suffit pas. Copier / coller les exemples ne suffit pas. Sinon, vous n'aurez que de vagues souvenirs de ce que vous lirez. Vous oublierez même que je vous avais averti que vous oublierez !

Présentation

Sur Internet, les protocoles de transport les plus utilisés sont TCP et UDP. Dans C++, on se sert de SOCK_STREAM et SOCK_DATAGRAM pour dire si un socket est en TCP ou en UDP. Lorsqu'on émet des données via un socket, le programme confie ces données au système d'exploitation qui les fait passer dans sa pile réseau. A chaque étape, il les encapsule convenablement avant de les envoyer sur le fil. Une pile réseau standard est :



Le raw socket est un moyen de court-circuiter la pile réseau du système d'exploitation. Charge au programme, donc au programmeur, donc à toi lecteur, de placer les bonnes entêtes convenablement remplies avant tes données. Si les entêtes ont des erreurs, la transmission ne se passera pas convenablement. Comme le disait une publicité pour les pneus, sans maîtrise la puissance n'est rien.

En émission, les raw sockets de Windows permettent au programmeur de laisser à la pile du système d'exploitation le soin de s'occuper de l'entête IP, ou bien de la prendre à sa charge également. En réception, un raw socket peut être configuré pour filtrer les paquets reçus afin de ne remettre au programme que les TCP, ou les ICMP, ou les IP. Ces deux mécanismes permettent de configurer les raw sockets de manière très souple, tant au niveau de ce qu'on désire envoyer qu'à celui de ce qu'on veut recevoir et traiter.

Les raw sockets n'ont pas bonne réputation, et on trouve des articles apocalyptiques comme http://grc.com/dos/winxp.htm. Il est vrai que, comme ne le disait pas une pub pour les pneus, sans puissance la maîtrise n'est rien. Pour bien du monde, connaître un domaine ne sert qu'à l'utiliser pour mal agir. Rendez-vous compte, à cause de gens comme ça Steve Gibson est devenu paranoïaque. Pour revenir au sujet, les raw sockets sont limités aux utilisateurs ayant les privilèges d'administration par les clé de registre HKLM\System\CurrentControlSet\Services\TcpIp\Parameters\AllowUserRawAccess et DisableRawSecurity.

On trouve également des implémentations tierces parties des raw sockets. Par exemple LibPCap (linux) / WinPCap (Windows). Cette librairie est très pratique et dispose de nombreuses possibilités en matière de tri des paquets reçus et de génération de statistique. Elle est à la base de nombreux sniffers comme Snort. De plus le passage d'un programme de Linux à Windows est facilité car l'interface programmeur est similaire. Cependant, actuellement elle souffre d'une limitation gênante. En effet elle n'est compatible qu'avec les interfaces réseau Ethernet, et ça n'est pas le cas pour les modems USB qui sont très répandus de nos jours.

I.1 - Analyse de TRACERT.EXE

Tracert.exe est le programme de traçage d'itinéraire de Microsoft. Il est fourni avec Windows, dans le répertoire System32. Ce programme affiche les adresses IP de tous les routeurs rencontrés entre nous et notre destination. Voici un exemple d'une sortie de tracert :

```
C:\>tracert -d www.yahoo.com

Détermination de l'itinéraire vers www.yahoo.akadns.net [66.94.230.32]
avec un maximum de 30 sauts :

  1   105 ms   113 ms   124 ms  193.253.160.3
  2    83 ms   229 ms    44 ms  80.10.207.130
  3   226 ms   130 ms   132 ms  193.252.101.26
  4    82 ms   273 ms    58 ms  193.252.103.110
  5    59 ms   177 ms    63 ms  193.252.161.57
  6    73 ms    76 ms    95 ms  193.251.126.58
  7    84 ms    87 ms   138 ms  193.251.243.29
  8   406 ms   468 ms   271 ms  193.251.243.170
  9   260 ms   138 ms   261 ms  193.251.243.82
 10   139 ms   321 ms   302 ms  193.251.243.194
 11   292 ms   149 ms   139 ms  193.251.254.126
 12   216 ms   216 ms   419 ms  216.115.98.49
 13   293 ms   296 ms   352 ms  216.115.101.42
 14   218 ms   388 ms   221 ms  66.218.82.230
 15   298 ms   263 ms   220 ms  66.94.230.32

Itinéraire déterminé.
```

En ouvrant tracert.exe au moyen du Dependancy Walker, un outil fourni avec VC++, on peut observer les bibliothèques utilisées par ce programme ainsi que les fonctions importées. Ca nous donnera une idée de son fonctionnement.

On y voit **WS2_32.DLL** avec les fonctions *GetAddrInfo*, *GetNameInfo* et *FreeAddrInfo*. Après un coup d'oeil sur www.msdn.microsoft.com, on sait que ces trois fonctions sont celles qui servent à convertir un nom d'hôte en adresse IP et inversement. Ces fonctions sons nommées GetHostByName et GetHostByAddr lorsqu'on les utilise classiquement avec Winsock.h. Dans l'encadré ci-dessus, le programme utilise GetNameInfo pour résoudre www.yahoo.com en son ip 66.94.230.32. Puis GetAddrInfo est ensuite appelé pour transformer cette ip en www.yahoo.akadns.net. Ces étapes sont les requêtes ARP (Adress Resolution Protocol) et RARP (Reverse ARP).

On y voit également **ICMP.DLL** avec *IcmpCreateFile*, *IcmpSendEcho2*, et *IcmpCloseHandle*. Un coup d'oeil sur l'aide en ligne confirmera notre intuition : on a mis dans le mille ! Ces trois fonctions permettent d'accéder à une sorte de socket pour la fonction Echo du protocole ICMP, c'est à dire le bien connu Ping. On a de quoi le créer, m'utiliser et le fermer proprement. Voici la définition de IcmpSendEcho2 :

```
DWORD IcmpSendEcho2(
  HANDLE IcmpHandle,
  HANDLE Event,
  FARPROC ApcRoutine,
  PVOID ApcContext,
  IPAddr DestinationAddress,
  LPVOID RequestData,
  WORD RequestSize,
  PIP_OPTION_INFORMATION RequestOptions,
  LPVOID ReplyBuffer,
  DWORD ReplySize,
  DWORD Timeout
);
```

Microsoft nous dit que IcmpSendEcho2 envoie une demande ICMP Echo, avec comme charge les données pointées par le champ *RequestData*. La réponse est stoquée dans un buffer. Si il n'y a pas de réponse dans le temps spécifié par le champ Timeout, on a une erreur. Le champ *RequestOption* est un pointeur vers d'éventuelles options IP à appliquer au paquet. Mais quelles options ?

```
typedef struct ip_option_information {
  UCHAR Ttl;
  UCHAR Tos;
  UCHAR Flags;
  UCHAR OptionsSize;
  PUCHAR OptionsData;
} IP_OPTION_INFORMATION, *PIP_OPTION_INFORMATION;
```

Le plus important pour nous est le champ *TTL*. Petit rappel sur le TTL. Lorsque j'utilise Internet Explorer pour aller lire la page d'accueil de Yahoo, j'émets un paquet à destination de www.yahoo.com. Ce paquet passe par la pile réseau de Windows. Cette pile met une valeur dans le champ TTL du protocole IP. A chaque fois que le paquet arrive sur un routeur, celui-ci décrémente le champ TTL et renvoie le paquet. Sauf si le TTL arrive à zéro. Dans ce cas le paquet est détruit, et le routeur me renvoie un paquet ICMP TIME EXCEEDED pour que je sois averti de cette destruction.

Le champ *ReplyBuffer* est une structure du type IP_ECHO_REPLY qui contient d'intéressantes informations. Qui a répondu ? En combien de temps ?

```
Typedef struct {
  DWORD Address;
  unsigned long  Status;
  unsigned long  RoundTripTime;
  unsigned short DataSize;
  unsigned short Reserved;
  void *Data;
  IP_OPTION_INFORMATION Options;
} IP_ECHO_REPLY, * PIP_ECHO_REPLY;
```

Le fonctionnement du programme se fait plus précis. J'envoie vers la destination des paquets ICMP Echo, tout en changeant la valeur du TTL, de 1 à autant que nécessaire. Pour chaque réponse je regarde quelle est l'IP qui me répond. La réponse est-elle un message de fin de vie de paquet ? Alors j'ai atteint un routeur dont je connais maintenant l'IP. Est-ce un ICMP Reply ? Dans ce cas j'ai atteint la destination. Fin du programme.

I.2 - Un traceroute avec ICMP.DLL

Voici la source d'un traceroute assez simple. Comme tracert.exe, il se base sur l'utilisation de ICMP.DLL. Mis à part la présentation, il fonctionne comme le tracert.exe de Microsoft.

Deux petites complications, mais pas grand chose. D'abord, ma fonction de ping inclut un timeout adaptatif pour optimiser les délais d'attente. Ca ne sert pas forcément à grand chose dans ce cas, mais je m'en étais servi dans le projet à partir duquel j'ai recopié des parties de code pour cet exemple. Je l'ai laissé, ca peut être intéressant à regarder. Ensuite, comme je n'ai pas trouvé de fichier .h pour les fonctions icmp, j'ai importé les fonctions de la DLL à la main. A part ça, tout est assez simple.

En regardant de plus près, je vois que de petites choses auraient besoin d'un lifting, notamment les moments où je veux afficher à l'écran des adresses ip. Des versions plus efficaces seront utilisées dans les exemples suivants.

```
#include "stdio.h"
#pragma comment (lib,"Ws2_32")
#include "winsock2.h"


  //Constantes---------------------------------------------------------------------------
    const nombre_de_hops_a_tracer = 25;


  //Types--------------------------------------------------------------------------------
    typedef struct
    {
      unsigned char Ttl;                          // Time To Live
      unsigned char Tos;                          // Type Of Service
      unsigned char Flags;                        // IP header flags
      unsigned char OptionsSize;                  // Size in bytes of options data
      unsigned char *OptionsData;                 // Pointer to options data
    } IP_OPTION_INFORMATION, * PIP_OPTION_INFORMATION;


    typedef struct
    {
      DWORD Address;                              // Replying address
      unsigned long  Status;                      // Reply status
      unsigned long  RoundTripTime;               // RTT in milliseconds
      unsigned short DataSize;                    // Echo data size
      unsigned short Reserved;                    // Reserved for system use
      void *Data;                                 // Pointer to the echo data
      IP_OPTION_INFORMATION Options;              // Reply options
    } IP_ECHO_REPLY, * PIP_ECHO_REPLY;


  //Pointeurs et handle de l'importation DLL---------------------------------------------
    HINSTANCE  hIcmp;
    typedef HANDLE (WINAPI* pfnHV)(VOID);
    typedef BOOL (WINAPI* pfnBH)(HANDLE);
    typedef DWORD (WINAPI* pfnDHDPWPipPDD)(HANDLE, DWORD, LPVOID, WORD,PIP_OPTION_INFORMATION, LPVOID, DWORD,
DWORD);
    pfnHV pIcmpCreateFile;
    pfnBH pIcmpCloseHandle;
    pfnDHDPWPipPDD pIcmpSendEcho;


  //Definition des fonctions-------------------------------------------------------------
    int do_the_ping (u_long ip_actuelle);
    int traite_ligne_de_commande (int argc, char** argv);
    int importe_icmp_from_dll ();


  //Variables globales-------------------------------------------------------------------
    //Timeout adaptatif
    int timeout_user = 3;
```

```
	int timeout_du_ping = 0;
	int accu_du_timeout = 0;
	int compteur_du_timeout=0;

	//IP de la ligne de commande
	u_long ip_cible;
	char conversion_ip_string[16];




//-----------------------------------------------------------------------------------------
// ***** MAIN *****
//-----------------------------------------------------------------------------------------
int main (int argc, char* argv[])
{
	//Initialise les services de WinSocks
	WSADATA wsadata;
	if (WSAStartup(MAKEWORD(1,1),&wsadata)!=0)
	   {printf("Erreur d'initialisation WINSOCS.\n");return -1;}

	//Initialise les services de Ping
	if (importe_icmp_from_dll () == -1)
	   {printf("Erreur d'initialisation ICMP.DLL.\n");return -1;}

	//Traite la ligne de commande
	if (traite_ligne_de_commande (argc, argv) == -1)
	   return -1;

	//Envoi du ping
	do_the_ping (ip_cible);

	// Libération de la DLL de PING
	FreeLibrary(hIcmp);

	//Libération des ressources WinSocks et fermeture
	WSACleanup();
	return 0;

}




//-----------------------------------------------------------------------------------------
// Traitement de la ligne de commande
//-----------------------------------------------------------------------------------------
int traite_ligne_de_commande (int argc, char** argv)
{
	// Vérifie le nombre de parametres
	if (argc < 2)
	{
	   printf ("usage: icmptrace <ip-debut>  [timeout]\n");
	   return -1;
	}

	// Recupere le parametre de timeout
	if (argc == 3)
	   timeout_user = atoi(argv[3]);
	timeout_du_ping = timeout_user*1000;

	//Vérifie l'IP donnée par la ligne de commande
	struct hostent* pHostEntConversion;

	//Résolution de nom
	pHostEntConversion = gethostbyname(argv[1]);
	if (pHostEntConversion == 0)
	{
	   printf ("Impossible de résoudre  %s/n", argv[1]);
	   return -1;
	}
	//Récupération de l'IP en String
	sprintf
```

```
    (
        conversion_ip_string,
        "%i.%i.%i.%i",
        (unsigned char)pHostEntConversion->h_addr_list[0][0],
        (unsigned char)pHostEntConversion->h_addr_list[0][1],
        (unsigned char)pHostEntConversion->h_addr_list[0][2],
        (unsigned char)pHostEntConversion->h_addr_list[0][3]
    );

    //Conversion de l'IP en numérique
    ip_cible = inet_addr(conversion_ip_string);
    if (ip_cible == INADDR_NONE)
    {
        printf ("L'IP %s est invalide/n", argv[1]);
        return -1;
    }

    return 1;
}




//----------------------------------------------------------------------------------------
// Importe les fonctions ICMP
//----------------------------------------------------------------------------------------
int importe_icmp_from_dll ()
{
    //Créé un pointeur sur la DLL
    hIcmp=LoadLibrary("ICMP.DLL");
    if (hIcmp == 0)
        {printf ("ICMP.DLL introuvable.\n");return -1;}

    //Importe les 3 fonctions depuis la DLL
    pIcmpCreateFile = (pfnHV)GetProcAddress(hIcmp,"IcmpCreateFile");
    pIcmpCloseHandle = (pfnBH)GetProcAddress(hIcmp,"IcmpCloseHandle");
    pIcmpSendEcho = (pfnDHDPWPipPDD)GetProcAddress(hIcmp,"IcmpSendEcho");

    return 1;
}




//----------------------------------------------------------------------------------------
// Lance la vague de ICMP ECHO REQUEST, traite les réponses
//----------------------------------------------------------------------------------------
int do_the_ping (u_long ip_actuelle)
{
  // 1) PREPARATION DU TERRAIN : L'ICMP
    //Créé un handle de Ping
    HANDLE hIP;
    hIP = pIcmpCreateFile();
    if (hIP == INVALID_HANDLE_VALUE)
        {printf("Impossible d'accéder aux services Ping.\n");return -1;}

    // Créé les structures du paquet : le Playload
    char acPingBuffer[64];
    memset(acPingBuffer, '\xAA', sizeof(acPingBuffer));

    // Créé les structures du paquet : le tampon pour le Reply
    PIP_ECHO_REPLY pIpe;
    pIpe = (PIP_ECHO_REPLY) malloc( sizeof(IP_ECHO_REPLY) + sizeof(acPingBuffer) );
    if (pIpe == NULL)
        {printf("Erreur d'allocation mémoire.\n");return -1;}
    memset (pIpe,0,sizeof(IP_ECHO_REPLY) + sizeof(acPingBuffer));

    // Créé les structures du paquet : le tampon pour les Optiond
    PIP_OPTION_INFORMATION pOptions;
    pOptions = (PIP_OPTION_INFORMATION) malloc ( sizeof(IP_OPTION_INFORMATION) );
    if (pOptions == NULL)
        {printf("Erreur d'allocation mémoire.");free(pIpe);return -1;}
    memset (pOptions,0,sizeof(IP_OPTION_INFORMATION));
```

```
    //Remplis les champs qui vont rester fixe tout au long de la procedure
    pIpe->Data = acPingBuffer;
    pIpe->DataSize = sizeof(acPingBuffer);


// 2) PREPARATION DU TERRAIN : DIVERSES VARIABLES LOCALES
    //Créé le HostEnt de la cible : numéro -> in_addr -> string -> hostent
    struct in_addr son_adresse_pour_string;
    struct hostent* pHostEntCible;
    son_adresse_pour_string.S_un.S_addr = ip_actuelle;
    pHostEntCible = gethostbyname(inet_ntoa(son_adresse_pour_string));


// 3) PING PRELIMINAIRE ET SON TRAITEMENT
    //Ping l'hôte pour détecter les Downs et les Firewallés
    printf("Ping préliminaire de %s\n",conversion_ip_string);
    DWORD dwStatus = pIcmpSendEcho
    (
        hIP,                                                //ICMP Handle
        *((DWORD*)pHostEntCible->h_addr_list[0]),           //Destination Adress
        acPingBuffer,                                       //Request Data
        sizeof(acPingBuffer),                               //Request Data Size
        NULL,                                               //Request Options
        pIpe,                                               //Reply Buffer
        sizeof(IP_ECHO_REPLY) + sizeof(acPingBuffer),       //Reply Buffer Size
        timeout_du_ping                                     //Timeout
    );

    //Analyse de la réponse
    if (dwStatus == 0)
        {printf("L'hôte ne répond pas au ping. Il est éteint ou derrière un firewall.\n");}
    else
    {
        char conversion_ip_string[16];
        sprintf(conversion_ip_string,"%i.%i.%i.%i",
            int(LOBYTE(LOWORD(pIpe->Address))),
            int(HIBYTE(LOWORD(pIpe->Address))),
            int(LOBYTE(HIWORD(pIpe->Address))),
            int(HIBYTE(HIWORD(pIpe->Address))));
        //Et que l'ip de réponse est l'ip de l'hote
        if (ip_actuelle != inet_addr(conversion_ip_string))
            {printf("L'hôte qui répond n'est pas l'hôte attendu.\n");}
        else
        {
            printf("L'hôte est valide.\n");

            //Calcul du timeout adaptatif
            compteur_du_timeout++;
            accu_du_timeout += pIpe->RoundTripTime;
            timeout_du_ping = timeout_user*(accu_du_timeout / compteur_du_timeout);


// 4) LANCE LA TRACE SUR CET HOTE QUI REPOND AU PING
            //Alors lance la Boucle de TraceRoute
            for (int i = 1;i<nombre_de_hops_a_tracer;i++)
            {
                //Règle le TTL
                memset (pOptions,0,sizeof(IP_OPTION_INFORMATION));
                pOptions->Ttl = i;

                // Envoi du paquet
                DWORD dwStatus = pIcmpSendEcho
                (
                    hIP,                                            //ICMP Handle
                    *((DWORD*)pHostEntCible->h_addr_list[0]),       //Destination Adress
                    acPingBuffer,                                   //Request Data
                    sizeof(acPingBuffer),                           //Request Data Size
                    pOptions,                                       //Request Options
                    pIpe,                                           //Reply Buffer
                    sizeof(IP_ECHO_REPLY) + sizeof(acPingBuffer),   //Reply Buffer Size
                    timeout_du_ping                                 //Timeout
                );

                if (dwStatus != 0)
```

```
            {
              //Ici on a un TTL Exceeded
              sprintf (conversion_ip_string,"%i.%i.%i.%i",
                 int(LOBYTE(LOWORD(pIpe->Address))),
                 int(HIBYTE(LOWORD(pIpe->Address))),
                 int(LOBYTE(HIWORD(pIpe->Address))),
                 int(HIBYTE(HIWORD(pIpe->Address))));
              printf("%s\n",conversion_ip_string);

              //Si on est arrivé à la destination voulue, on passe au suivant
              if (inet_addr(conversion_ip_string) == ip_actuelle)
                    break;
            }//fin du if le status est bon
            else
            {
              // Ici on a un time out
              printf("X.X.X.X\n");
            }//fin du else le status est mauvais

          }//fin de la boucle de trace
       }

     }//fin du Else d'esquive d'hote down et de firewall

     free(pIpe);
     return 1;
}//fin de la fonction
```

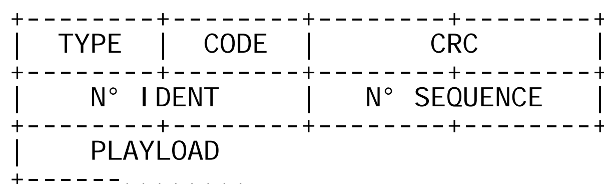
II.1 - Les entêtes dans la tête - ICMP

Le traceroute de Microsoft est basé sur des Pings, c'est à dire des ICMP ECHO REQUEST. Commencer à aborder les raw sockets par ICMP me semble une bonne idée. On a vu qu'avec les raw sockets c'est au programmeur de gérer les entêtes des protocoles. Ca va donc être à nous de gérer ICMP (Internet Control Message Protocol).

A l'émission d'un ICMP ECHO REQUEST on peut avoir plusieurs réponses possibles :

- ICMP ECHO REPLY lorsque tout va bien.
- ICMP TIME EXCEEDED lorsque le TTL est insuffisant. C'est le routeur qui met à mort le paquet qui envoie la réponse.
- ICMP HOST UNREACHABLE lorsque la cible n'est pas en ligne, ou lorsqu'un firewall la masque.
- Un time out si la réponse est trop longue à arriver ou si un firewall détruit notre paquet ICMP sans y répondre.

Voici la structure d'une entête ICMP. Elle est suivie d'un playload de données au choix de l'utilisateur.



La valeur du type définit le type du paquet :

ICMP ECHO REQUEST	8
ICMP ECHO REPLY	0
ICMP TIME EXCEEDED	11
ICMP HOST UNREACHABLE	3

Pour notre utilisation, le code ne va pas servir. Il est utilisé dans d'autres types de paquets ICMP. Les numéros de séquence et d'identifications sont libres. Ils servent au programmeur à bien faire correspondre un ECHO REPLY à un ECHO REQUEST. Il est courant de mettre le numéro de processus dans Ident. Comme les numéros seq et id ne servent qu'au programmeur, il n'est pas nécessaire d'utiliser de ntoh et hton pour les manipuler. Je parlerais plus de ntoh et hton dans le chapitre III. Le playload de la réponse doit être le même que le playload de la requête.

Attention, dans les paquets TIME EXCEEDED et HOST UNREACHABLE, les numéros de séquence et d'identification ne contiennent pas du tout les valeurs du paquet ECHO REQUEST. Le playload est différent aussi car il contient en fait l'entête IP et les 64 premiers bits (8 octets) du paquet qui est à l'origine de l'erreur.

Reste le CRC. On retrouve un CRC dans la plupart des entêtes de protocoles. Dans certains cas, comme pour ICMP, le CRC est calculé sur l'entête plus le playload concerné. Dans d'autres cas, comme IP, le CRC n'est calculé que sur l'entête. Tout dépend de si le protocole a pour vocation de garantir l'intégrité des données, ou juste de son entête à lui. IP ne protège que l'entête IP alors que TCP assure à l'utilisateur que les données transmises ne sont pas corrompues. Si et TCP et IP calculaient un CRC sur l'intégralité de la trame, ca ferait pas mal d'opérations inutiles. ICMP, comme TCP, garantit que le playload n'est pas corrompu. Dans tous les cas, la fonction de calcul de CRC est la même, seuls les paramètres changent. Avant le calcul du CRC, le champ CRC de l'entête doit être mis à zéro.

```
__int16 crc (__int16* addr, int count)
{
   __int32 sum = 0;
   while( count > 1 )
      {sum += *addr++;count -= 2;}

   if(count)
      sum += * (__int8 *) addr;

   while (sum>>16)
      sum = (sum & 0xffff) + (sum >> 16);

   return (__int16) ~sum;
}
```

Voici par exemple un ICMP ECHO REQUEST contient le playload COCO. Il est parfaitement valide et prêt à être "CRCisé" :

```
+--------+--------+--------+--------+
|   8    |   0    |        0        |
+--------+--------+--------+--------+
|     123456      |     1658734     |
+--------+--------+--------+--------+
|   C    |   O    |   C    |   O    |
+--------+--------+--------+--------+
```

On obtiendra le paquet ICMP ECHO REPLY suivant si il n’y a pas de problème de TTL :

```
+--------+--------+--------+--------+
|   0    |   0    |        0        |
+--------+--------+--------+--------+
|     123456      |     1658734     |
+--------+--------+--------+--------+
|   C    |   O    |   C    |   O    |
+--------+--------+--------+--------+
```

Nous voilà familiarisés avec les structures ICMP. Maintenant, voici comment déclarer le raw socket dont je vais me servir. En le spécifiant comme SOCK_RAW, je préviens que je vais donner les entêtes moi-même. Par défaut, Windows garde le contrôle de l’entête IP, ce qui est exactement ce que je souhaite. En spécifiant le socket comme étant IPPROTO_ICMP, je règle le socket de telle sorte que lorsque j’écouterais dessus, seuls les paquets ICMP me seront délivrés. J’évite ainsi la tâche de trier tous les paquets reçus par mon ordinateur, à la recherche des paquets ICMP qui m’intéressent.

```
sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

Par contre je ne peux plus régler le TTL comme lorsque j’utilisais ICMP.DLL. Au lieu de ça, j’utilise la fonction SetSockOpt qui, comme son nom l’indique, sert à régler diverses options du socket. Je dis vouloir régler des options concernant l’entête IP, et plus particulièrement le champ TTL. La valeur que je donne en paramètre sera alors insérée convenablement dans l’entête IP par la pile réseau lorsque Windows émettra la trame sur le fil.

```
setsockopt(sd, IPPROTO_IP, IP_TTL, (const char*)&valeur_ttl, sizeof(valeur_ttl)) ;
```

L’émission sur un raw socket se passe généralement en mode non connecté , au moyen de la fonction SendTo. Cela demande à ce qu’une structure SockAddr soit correctement initialisée pour désigner la cible voulue. Le champ Port du SockAddr est négligeable, car il est utilisé dans le cas de TCP et d’UDP. Le protocole ICMP n’a pas de ports.

```
hostent* hp;
sockaddr_in dest;

memset(&dest, 0, sizeof(dest));

hp = gethostbyname(argv[1]);
memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
dest.sin_family = hp->h_addrtype;
```

Une fois cette structure renseignée, il n'y a plus qu'à émettre. La pile réseau recopiera l'adresse fournie par le sockAddr dans l'entête IP.

```
taille = sendto (sd, (char*)send_buf, packet_size, 0, (sockaddr*)&dest, sizeof(dest));
```

Et voici pour la réception.

```
int taille  = sizeof(source);
sockaddr_in source;

recvfrom (sd, (char*)recv_buf, MAX_PING_PACKET_SIZE + sizeof(IPHeader), 0,(sockaddr*)&source, &taille);
```

L'utilisation du SockAddr dans la réception n'est pas une nécessité. Les deux derniers paramètres sont remplaçables par NULL. Mais dans cet exemple, j'utilise cette possibilité. Pour afficher l'IP du poste qui envoie une réponse, quelle qu'elle soit, il me suffit de consulter le SockAddr source.

Pour essayer d'etre le plus complet possible, un petit mot sur l'absence de Bind. Le programme ne se s'en sert pas car le socket est implicitement lié à mon ip lors de sa première utilisation en émission. RecvFrom sait donc sur quelle interface utiliser le socket lors de l'écoute. Même si vous utilisez RecvFrom sans spécifier de SockAddr, le bind n'est pas nécessaire.

II.2 – Utilité de la manipulation ICMP

Le protocole ICMP est un canal de communication souvent détourné de son utilisation dans le hack. Les pings sont souvent un préliminaire à une attaque en profondeur. De même, des refus de service peuvent être lancés au moyen de rafales de pings de grande taille.

ICMP peut également servir de *canal de communication caché*. Un troyen installé sur un ordinateur et surveillant le traffic ICMP pour y trouver ses commandes n'ouvre pas de port, ce qui peut être un grand avantage. Le troyen ne peut pas se repérer en scannant l'ordinateur, donc le hacker sait que son forfait n'est pas visible de l'exterieur. Un scan de port par un script d'audit PHP ne révèlera rien d'anormal, et un script kiddie scannant à gogo afin de trouver des ordinateurs offrant un accès via trojan ne le trouvera pas non plus. Sur le poste même, un *netstat –an* ne révèlera pas d'ouverture de port suspecte. Le premier avantage de l'ICMP est donc de ne pas être détecté par les logiciels listant les connexions TCP / UDP actives.

ICMP peut s'avérer efficace pour leurer certains firewalls. Si des règles de sécurité sont très agressives envers TCP et UDP mais que le protocole ICMP est autorisé à traverser le firewall, alors un troyen icmp percera efficacement une telle protection. Un angle d'attaque de plus pour chercher une voie d'accès. Le second avantage de l'ICMP est de ne pas dépendre des mêmes règles de sécurité que TCP / UDP.

Dans le cas d'une telle utilisation, ICMP ressemble à UDP dans le sens où il n'y a pas de couple SEQ – ACK qui garantisse le bon acheminement de tous les paquets. Par exemple, un logiciel de transfer de fichier via ICMP nécessite des vérifications à la charge du programmeur, sinon rien ne garantit que tous les fragments de fichier soient arrivés, et que ceux qui ont été reçus soient dans le bon ordre. Le grand défaut d'ICMP est que le programme doit gérer lui-même l'acquitement et la réorganisation des paquets.

II.3 - Un traceroute en raw sockets ICMP

Le code est simple mais pas forcément présenté au mieux. Je sais qu'un programme gagne à être présenté en fonctions. Du point de vue de son auteur, en tout cas. Pour ma part, quand je lis un code source pour le comprendre, j'aime assez peu passer de la page 1 à la page 42 en recherchant le code qui correspond à telle ou telle fonction pour comprendre ce que fait le programme. C'est pourquoi le code n'est pas écrit de manière fonctionnelle, mais de manière linéaire comme un article de journal. Seule la fonction de checksum est isolée à la fin du source.

```
#include <winsock2.h>
#pragma comment (lib,"Ws2_32")
#include <iostream.h>
#include <stdio.h>
#include <ws2tcpip.h>

#define DEFAULT_PACKET_SIZE 32
#define DEFAULT_TTL 30
#define MAX_PING_DATA_SIZE 1024
#define MAX_PING_PACKET_SIZE (MAX_PING_DATA_SIZE + sizeof(IPHeader))

#define ICMP_ECHO_REPLY 0
#define ICMP_DEST_UNREACH 3
#define ICMP_TTL_EXPIRE 11
#define ICMP_ECHO_REQUEST 8


// The IP header
typedef struct  {
   BYTE h_len:4;          // Length of the header in dwords
   BYTE version:4;        // Version of IP
   BYTE tos;              // Type of service
   USHORT total_len;      // Length of the packet in dwords
   USHORT ident;          // unique identifier
   USHORT flags;          // Flags
   BYTE ttl;              // Time to live
   BYTE proto;            // Protocol number (TCP, UDP etc)
   USHORT checksum;       // IP checksum
   ULONG source_ip;
   ULONG dest_ip;
}IPHeader, *pIPHeader;

// ICMP header
typedef struct {
   BYTE type;         // ICMP packet type
   BYTE code;         // Type sub code
   USHORT checksum;
   USHORT id;
   USHORT seq;
}ICMPHeader, *pICMPHeader;

USHORT ip_checksum(USHORT* buffer, int size);


int main(int argc, char* argv[])
{
   // VARIABLES : les buffers
      pICMPHeader send_buf = NULL;
      pIPHeader recv_buf = NULL;

   // VARIABLES : pour la destination
      sockaddr_in dest, source;
      hostent* hp;

   // VARIABLES : pour le ping
      int seq_no = 10;
      SOCKET sd;
      int packet_size = DEFAULT_PACKET_SIZE;
      int ttl = DEFAULT_TTL;
      int taille;

      int i;
```

```
// ETAPE 1 : GERER LA LIGNE DE COMMANDE
  // Message d'erreur en cas de manque de paramètres
    if (argc < 2)
    {
      printf("usage: rawping <host> [taille donnees] [ttl]\n");
      printf ("\tles donnees peuvent aller jusqu'a %i bytes. Par defaut %i\n",MAX_PING_DATA_SIZE,DEFAULT_PACKET_SIZE);
      printf("\tle ttl peut aller jusqu'a 255.  Par default %i\n",DEFAULT_TTL );
      exit (1);
    }

  // Traite les paramètres optionnels
    if (argc > 2)
    {
      int temp = atoi(argv[2]);
      if (temp != 0)
        packet_size = temp;

      if (argc > 3)
      {
        temp = atoi(argv[3]);
        if ((temp >= 0) && (temp <= 255))
          ttl = temp;
      }
    }

  // Borne la taille du paquet
    if (packet_size > MAX_PING_DATA_SIZE)
      packet_size = MAX_PING_DATA_SIZE;
    if (packet_size < sizeof(ICMPHeader))
      packet_size = sizeof(ICMPHeader);



// ETAPE 2 : INITIALISATION DES RESSOURCES RESEAU
  // Lance WinSocks
    WSAData wsaData;
    if (WSAStartup(MAKEWORD(2, 1), &wsaData) != 0)
    {
      printf("Impossible de démarrer WinSock. Version peut-être inférieure à 2.1\n");
      return 1;
    }


  // Créé le socket
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sd == INVALID_SOCKET)
    {
      printf("Erreur de création du Raw Socket : %s\n",WSAGetLastError() );
      goto cleanup;
    }


  // S'occupe du Sockaddr
    memset(&dest, 0, sizeof(dest));
    hp = gethostbyname(argv[1]);
    if (hp != 0)
    {
      memcpy(&(dest.sin_addr), hp->h_addr, hp->h_length);
      dest.sin_family = hp->h_addrtype;
    }
    else
    {
      printf("Impossible de résoudre %s\n",argv[1]);
      goto cleanup;
    }

  // Alloue les buffers
    send_buf = (ICMPHeader*)malloc(packet_size);
    if (send_buf == NULL)
    {
      printf("Impossible d'allouer le buffer d'emission\n");
      goto cleanup ;
    }
```

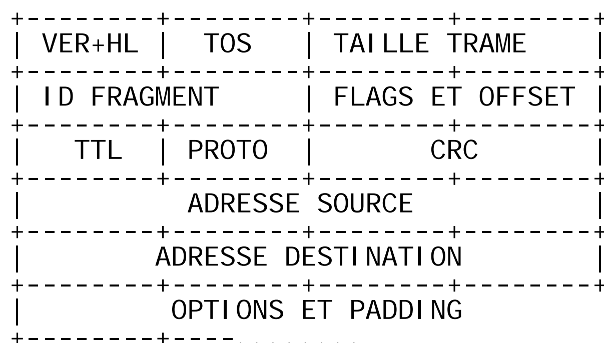
```
        recv_buf = (IPHeader*)malloc(MAX_PING_PACKET_SIZE);
        if (recv_buf == NULL)
        {
           printf("Impossible d'allouer le buffer de reception\n");
           goto cleanup;
        }


//Boucle de trace
for (i = 1; i < ttl ;i++)
{

   // ETAPE 3 : INITIALISATION DES RESSOURCES RESEAU
      // Règle le paquet ICMP
        send_buf->type = ICMP_ECHO_REQUEST;
        send_buf->code = 0;
        send_buf->checksum = 0;
        send_buf->id = (USHORT)GetCurrentProcessId();
        send_buf->seq = i;

      // Et remplit le playload libre après le timestamp
        {
           char* datapart = (char*)send_buf + sizeof(ICMPHeader);
           int bytes_left = packet_size - sizeof(ICMPHeader);
           while (bytes_left > 0)
           {
              memcpy(datapart, "A", 1);
              bytes_left--;
              datapart++;
           }
        }

      // Calcule le CRC
        send_buf->checksum = ip_checksum((USHORT*)send_buf, packet_size);


   // ETAPE 4 : ACTION !!!
      // Envoi du ping, avec le ttl voulu

        // Règle son TTL dans les options IP
        if (setsockopt(sd, IPPROTO_IP, IP_TTL, (const char*)&i, sizeof(i)) == SOCKET_ERROR)
        {
           printf("Impossible de régler le ttl : %s\n",WSAGetLastError() );
           goto cleanup;
        }

        taille = sendto(sd, (char*)send_buf, packet_size, 0, (sockaddr*)&dest, sizeof(dest));
           if (taille == SOCKET_ERROR)
        {
           printf("Erreur d'emission\n");
           goto cleanup;
        }


      // Boucle de réception
        bool recu = false;
        while (!recu)
        {

           //Ecoute
           taille  = sizeof(source);
           taille = recvfrom(sd, (char*)recv_buf, MAX_PING_PACKET_SIZE + sizeof(IPHeader), 0,(sockaddr*)&source, &taille);
           if (taille == SOCKET_ERROR)
           {
              printf("Erreur de réception\n");
              goto cleanup;
            }

           if (taille <  packet_size + sizeof(IPHeader))
              printf("Erreur");


           //Va sur l'entete icmp
            ICMPHeader* icmphdr = (ICMPHeader*) ((char*)recv_buf + (recv_buf->h_len * 4));
```

```
            //Teste les différents cas
            if (icmphdr->type == ICMP_TTL_EXPIRE)
            {
               IPHeader* iphdr=(IPHeader *)icmphdr+sizeof(ICMPHeader);
               ICMPHeader* icmphdr2 = (ICMPHeader*) ((char*)iphdr + (iphdr->h_len * 4));

               printf("%s\n",inet_ntoa(source.sin_addr));
               recu = true;
            }

            else if (icmphdr->type == ICMP_ECHO_REPLY)
            {
               if (icmphdr->seq != i)
                  continue;

               printf("%s\n",inet_ntoa(source.sin_addr));
               recu = true;
               i = ttl;
            }

            else if (icmphdr->type == ICMP_DEST_UNREACH)
            {
               printf("x.x.x.x\n");
               recu = true;
            }
          }

}

cleanup:
   free(send_buf);
   free(recv_buf);
   WSACleanup();
   return 0;
}



u_short ip_checksum (u_short* buffer, int size)
{
   u_long sum = 0;
   while( size > 1 )      {sum += *buffer++;size -= 2;}

   if(size)         sum += * (u_char *) buffer;

   while (sum>>16)                  sum = (sum & 0xffff) + (sum >> 16);

   return (u_short) ~sum;
 }
```

III.1 - Les entêtes dans la tête - IP

Sous ICMP, on trouve la couche réseau IP (Internet Protocol). Celle-ci sert à gérer la liaison entre l'ordinateur source et l'ordinateur destination au moyen des bien connues adresses IP. L'entête comporte des champs servant à paramétrer cette liaison et le routage. C'est grâce à IP qu'une trame de grande taille peut être fragmentée en plusieurs petites trames, et être recomposée convenablement après. IP gère également le TTL, qui sert à ce qu'un paquet mal routé n'erre pas éternellement sur le réseau.

IP comporte également le numéro de version (ver), qui vaut 4 encore de nos jours car IPV6 est loin d'avoir relégué IPV4 aux oubliettes. La taille de l'entête (hl) est spécifiée, exprimée en mots de 32 bits. Des options peuvent être ajoutées, faisant ainsi varier cette taille. Sans options, la taille vaut 5. La taille du paquet est la taille totale de la trame, entête comprise. On trouve également le numéro de protocole de nuveau supérieur, qui vaudra 1 pour indiquer ICMP dans cet exemple. Voici un plan de l'entête IP :



```
+--------+--------+--------+--------+
| VER+HL |  TOS   | TAILLE TRAME    |
+--------+--------+--------+--------+
| ID FRAGMENT     | FLAGS ET OFFSET |
+--------+--------+--------+--------+
|  TTL   | PROTO  |      CRC        |
+--------+--------+--------+--------+
|         ADRESSE SOURCE            |
+--------+--------+--------+--------+
|       ADRESSE DESTINATION         |
+--------+--------+--------+--------+
|        OPTIONS ET PADDING
+--------+----........
```

Pour le moment, je ne vais pas m'étendre plus sur l'entête IP. Mon but, ca n'est pas de procéder à la façon d'un dictionnaire et balancer plein de choses sans qu'on s'en serve. Avec ce qu'on a vu, on en a assez pour créer des entêtes IP adaptées à nos besoins plutôt simples. Pour plus de détails sur la fragmentation ou les options, un petit tour par la RFC ne fera pas de mal.

Quelques précautions sont à prendre. Les champs de plus de 8 bits sont en Network Byte Order. La taille de la trame, par exemple, ets à stoquer apès l'avoir retournée avec htons (host to network – short). Les adresses sources et destinations sont également à manipuler avec précaution. Les fonctions de WinSock fonctionnent généralement toutes en network byte order, donc les conversions sont inutiles. Par exemple, inet_addr renvoie un entier long non signé en network byte order, tout prêt à être utilisé dans une entête protocolaire, dans un sockaddr, dans un in_addr.

Nous savons déclarer un raw socket filtré de type ICMP. Mais dans l'exemple précédent, ce socket n'était pas si brut que ça car l'entête IP était fabriquée par Windows. Il faut indiquer au Socket que c'est nous qui allons fournir l'entête IP. Dans l'exemple précédent, on a déjà eu l'occasion de faire des réglages sur un socket, c'était pour régler le TTL. Cette fois, on règle l'option IP_HDRINCL qui indique que l'entête IP est fournie par le programme. IP_HDRINCL est défini dans ws2tcpip.h.

```
BOOL hdrincl = 1;

sd = socket (AF_INET, SOCK_RAW, IPPROTO_ICMP);
setsockopt (sd, IPPROTO_IP, IP_HDRINCL, (const char*)&hdrincl, sizeof(hdrincl)) ;
```

Ensuite, nous réglons les champs IP et ICMP, en calculant convenablement les CRC, et on envoie la sauce. Pour l'écoute, il n'y a pas de grand changement. En effet, IP_HDRINCL concerne uniquement l'envoi sur le socket . Dans l'exemple précédent, écouter sur un raw socket nous retournait déjà le paquet avec son entête IP.

Un changement intervient pour la gestion du timeout. Je fais passer le socket en mode débloqué, et je teste la réception dans une boucle. Ceci est un des moyens pour bénéficer d'un timeout de la durée que l'on désire lors de l'écoute d'un socket.

```
unsigned long son_masque_ioctlsocket = 1;
```

```
ioctlsocket (sd, FIONBIO, &son_masque_ioctlsocket) ;
```

Lors de l'appel à recvfrom, comme évoqué au chapitre II.1, je n'utilise plus de sockaddr. L'adresse IP de l'ordinateur qui me renvoie les ICMP sera lue directement dans l'entête IP de ses trames.

Il y a une autre gymnastique dans le programme, pour obtenir l'adresse IP locale qui sera placée dans l'entête IP. Sur mon pc j'ai un modem ADSL USB et une carte ethernet. Résoudre mon nom en adresses me donne donc deux adresses IP possible. Il a donc été nécessaire de développer une petite routine pour déterminer laquelle est la plus probable d'être celle face à internet. D'autres méthodes sont envisageables, celle-ci marche bien dans mon cas.

Une remarque quand à l'émission sur le socket. Comme nous gérons nous-même l'entête IP, nous y mettons l'adresse de destination que l'on désire. SendTo ne se sert donc plus du SockAddr pour y lire cette adresse et l'insérer dans l'entête. Bien qu'il soit nécessaire de mettre un SockAddr dans SendTo, on peut y mettre n'importe quelle adresse IP, seule celle qu'on insère nous-même dans l'entête IP compte. Vous pouvez mettre l'IP de Google dans l'entête et utiliser un SockaAddr qui désigne Lycos, vous tracerez bel et bien Google.

```
taille = sendto(sd, (char*) send_buf, global_size, 0, (sockaddr*)&sa_dest, sizeof(sa_dest));
```

III.2 – Utilité de la manipulation IP

La manipulation directe de l'entête ICMP permet deux principales choses : jouer avec la fragmentation, et falsifier l'adresse IP source.

La falsification de l'IP source est le Spoofing. Si l'adresse IP source est spoofée, les firewalls, sniffers, la cible même penseront que cette fausse IP est bel et bien l'initiatrice de la communication. Par contre c'est à cette fausse IP que les réponses seront envoyées, et non pas à l'attaquant.

On peut par exemple envoyer un paquet en petits fragments :
A – B – C – D – E – X

L'ordinateur cible réceptionne un à un les fragments et recompose le paquet :
A
AB
ABC
ABCD
ABCDE
ABXCD

Le fragment X est concu pour venir écraser un bout déjà existant dans le paquet, modifiant par exemple l'adresse IP d'origine lorsqu'elle a été vérifiée par le firewall qui protège la cible. Cette méthode est utilisée par Teardrop.

Ping Of Death créé un paquet ICMP ECHO de taille supérieure au maximum permis par IP, et le fragmente en de nombreux petits bouts. Sur la cible, ces petits bouts sont recolés pour former ce paquet invalide qui cause le plantage du système.

III.3 - Un traceroute en raw sockets IP - ICMP

```
#include <winsock2.h>
#pragma comment (lib,"Ws2_32")
#include <stdio.h>
#include <ws2tcpip.h>
#include <winbase.h>



// Concernant ICMP-----------------------------------------------------------------------
  #define ICMP_ECHO_REPLY              0
  #define ICMP_DEST_UNREACH            3
    #define ICMP_DEST_UNREACH_NET_UNREACH                          0
    #define ICMP_DEST_UNREACH_HOST_UNREACH                         1
    #define ICMP_DEST_UNREACH_PROTOCOL_UNREACH                     2
    #define ICMP_DEST_UNREACH_PORT_UNREACH                         3
    #define ICMP_DEST_UNREACH_FRAG_NEEDED_DF                       4
    #define ICMP_DEST_UNREACH_SOURCE_ROUTE_FAILED                  5
  #define ICMP_SOURCE_QUENCH           4
  #define ICMP_REDIRECT                5
    #define ICMP_REDIRECT_NETWORK                   0
    #define ICMP_REDIRECT_HOST                      1
    #define ICMP_REDIRECT_TOS_NETWORK               2
    #define ICMP_REDIRECT_TOS_HOST                  3
  #define ICMP_ECHO_REQUEST            8
  #define ICMP_TIME_EXPIRE             11
    #define ICMP_TIME_EXPIRE_TTL                              0
    #define ICMP_TIME_EXPIRE_FRAG_REASSEMBLY_EXPIRE           1
  #define ICMP_PARAM_PROBLEM           12
    #define ICMP_PARAM_PROBLEM_POINTER_SET                    0
  #define ICMP_TIMESTAMP_REQUEST 13
  #define ICMP_TIMESTAMP_REPLY         14
  #define ICMP_INFO_REQUEST            15
  #define ICMP_INFO_REPLY              16

  // ICMP header
  typedef struct
  {
    BYTE type;
    BYTE code;
    USHORT checksum;
    USHORT id;
    USHORT seq;
  } ICMPHeader, *pICMPHeader;




// Concernant IP--------------------------------------------------------------------------
  #define IP_PROTO_ICMP       1
  #define IP_PROTO_TCP        6
  #define IP_PROTO_UDP        17

  // The IP header
  typedef struct
  {
    BYTE ihl:4;
    BYTE version:4;
    BYTE tos;
    USHORT total_len;
    USHORT ident;
    USHORT flags;
    BYTE ttl;
    BYTE protocol;
    USHORT checksum;
    ULONG source_ip;
    ULONG dest_ip;
  } IPHeader, *pIPHeader;
```

```
   #define ICMP  SIZE 32
   #define MAX_TTL 30
   #define MAX_PING_PACKET_SIZE (1024 + sizeof(IPHeader))
   #define RETRY_LOOP 10
   #define RETRY_WAIT 200

   USHORT ip_checksum(USHORT* buffer, int size);


int main(int argc, char* argv[])
{
   // VARIABLES : les buffers
      pIPHeader send_buf          = NULL;
      pIPHeader send_ip           = NULL;
      pICMPHeader send_icmp       = NULL;
      pIPHeader recv_buf          = NULL;

   // VARIABLES : pour la destination et la source
      sockaddr_in sa_dest;
      hostent* he_resolu;
      ULONG local,distant;

   // VARIABLES : pour le ping
      SOCKET sd;

   // COMPTEUR pour le trace
      int ttl_en_cours;

   // DIVERS pour le réglage du socket et sa taille
      int global_size;


   // ETAPE 1 : GERER LA LIGNE DE COMMANDE
      // Message d'erreur en cas de manque de paramètres
         if (argc < 1)
            {printf("usage: rawping <host> \n");exit (1);}

         global_size = ICMP_SIZE+sizeof(IPHeader);


   // ETAPE 2 : INITIALISATION DES RESSOURCES RESEAU
      // Lance WinSocks
         WSAData wsaData;
         if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
            {printf("Impossible de démarrer WinSock. Version peut-être inférieure à 2.1\n");return 1;}

      // Alloue les buffers. Le source est indexé convenablement
         send_buf = (pIPHeader) malloc(global_size);
         if (send_buf == NULL)
            {printf("Impossible d'allouer le buffer d'emission\n");goto cleanup;}
         memset(send_buf,0,global_size);
         send_ip =  (pIPHeader)            send_buf;
         send_icmp = (pICMPHeader)         ( (char*)send_buf+sizeof(IPHeader) );

         recv_buf = (pIPHeader)malloc(MAX_PING_PACKET_SIZE+sizeof(IPHeader));
         if (recv_buf == NULL)
            {printf("Impossible d'allouer le buffer de reception\n");goto cleanup;}


      //Calcule les adresses locales et distantes
         //La distante depuis argv[1]
         he_resolu = gethostbyname(argv[1]);
         distant = *(ULONG*)he_resolu->h_addr_list[0];

         //Calcul de la meilleure IP locale
         {
            //Chope mon ip : résolution de mon nom
            char hostname[1024];
            gethostname(hostname,1024);
            he_resolu = gethostbyname(hostname);


            //Recherche de l'ip la plus basse
            int laquelle_est_bonne = 0;
            unsigned char valeur = 255;
```

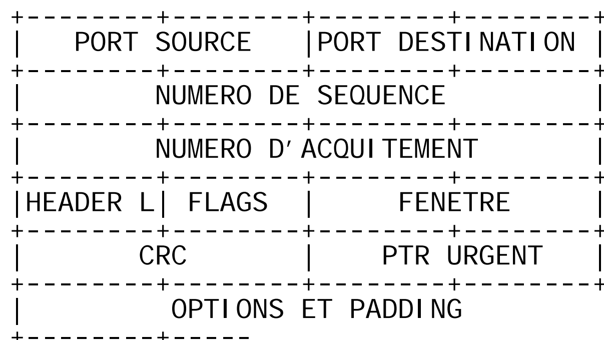
```
            int i=0;
            while (he_resolu->h_addr_list[i] != 0)
            {
               if (*(unsigned char*)he_resolu->h_addr_list[i] < valeur)
               {
                  laquelle_est_bonne = i;
                  valeur = *(unsigned char*)he_resolu->h_addr_list[i];
               }
               i++;
            }

            local = *(ULONG*)he_resolu->h_addr_list[laquelle_est_bonne];
         }

      //Créé le sockaddr distant pour l'émission
         memset(&sa_dest,0,sizeof(sockaddr));
         sa_dest.sin_family = AF_INET;
         sa_dest.sin_port = 0;
         *(ULONG*)&sa_dest.sin_addr = distant;

      //Affiche l'état actuel
         printf("résoultion de %s en %s\n",argv[1], inet_ntoa( *(in_addr*) &distant) );
         printf("mon ip est : %s\n\n",inet_ntoa( *(in_addr*) &local) );

      // Créé le RAW SOCKET ICMP, le règle en IP HEADER inclus et non bloquant
         {
            BOOL hdrincl = 1;
            unsigned long son_masque_ioctlsocket = 1;

            sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
            if (sd == INVALID_SOCKET)
               {printf("Erreur de création du Raw Socket : %s\n",WSAGetLastError() ); goto cleanup;}

            if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (const char*)&hdrincl, sizeof(hdrincl)) == SOCKET_ERROR)
               {printf("Impossible de régler le header ip inclus : %s\n",WSAGetLastError() );goto cleanup;}

            if (ioctlsocket (sd, FIONBIO, &son_masque_ioctlsocket)!=0)
               {printf("Impossible de débloquer le socket : %s\n",WSAGetLastError() );goto cleanup;}
         }



//////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////
//                    //
// LA BOUCLE DE TRACE //
//                    //
////////////////////////
      for (ttl_en_cours = 1; ttl_en_cours < MAX_TTL ;ttl_en_cours++)
      {

   // ETAPE 3 : REGLE LES ENTETES
      //S'occupe de l'entête IP
         send_ip->version = 4;
         send_ip->ihl = 5;
         send_ip->tos = 0;
         send_ip->total_len = htons(global_size);
         send_ip->ident = 0;
         send_ip->flags = 0;
         send_ip->ttl = ttl_en_cours;
         send_ip->protocol = IP_PROTO_ICMP;
         send_ip->checksum = 0;
         send_ip->source_ip = local;
         send_ip->dest_ip =   distant;
         send_ip->checksum = ip_checksum((USHORT*)send_ip, sizeof(IPHeader));

   // Règle la partie ICMP
         send_icmp->type = ICMP_ECHO_REQUEST;
         send_icmp->code = 0;
         send_icmp->checksum = 0;
         send_icmp->id =  ttl_en_cours;
         send_icmp->seq = ttl_en_cours;
   // Et remplit le playload libre après le timestamp
         memset (   ( (char*)send_icmp + sizeof(ICMPHeader) ), 'A', ICMP_SIZE - sizeof(ICMPHeader)   );
```

```
  // Calcule le CRC
        send_icmp->checksum = ip_checksum((USHORT*)send_icmp, ICMP_SIZE);




  // ETAPE 4 : ACTION !!
        int taille;
        int attente;

     // Envoi du ping, avec le ttl voulu
        taille = sendto(sd, (char*) send_buf, global_size, 0, (sockaddr*)&sa_dest, sizeof(sa_dest));
        if ( (taille == SOCKET_ERROR) | (taille != global_size) )
           {printf("Erreur d'emission %i\n", WSAGetLastError());goto cleanup;}


     // Boucle de réception
        bool recu = false;
        while (!recu)
        {

           //Ecoute
           memset(recv_buf,0,MAX_PING_PACKET_SIZE+sizeof(IPHeader));
           taille = 0;

           for (attente=0; attente<RETRY_LOOP; attente++)
           {
              //Ecoute sur le socket
              taille = recvfrom(sd, (char*)recv_buf, MAX_PING_PACKET_SIZE + sizeof(IPHeader), 0,0,0);

              //Si erreur
              if (taille == SOCKET_ERROR)
              {
                 //Et que besoin de plus de temps
                 if (WSAGetLastError()==WSAEWOULDBLOCK) //WSAEWOULDBLOCK = 10035
                    //on attend
                    Sleep(RETRY_WAIT);
                 else
                 {
                    //si l'erreur est grave
                    printf("Erreur de réception\n");
                    goto cleanup;
                 }
              }//fin du si erreur

              //Si pas d'erreur et taille non nulle on a assez entendu
              else if (taille != 0)
                 break;
           }




           //Si l'écoute a causé une erreur, on affiche X et on passe au hop suivant
           if (taille <= 0)
           {
              printf("x.x.x.x\n");
              break;
           }


           //Règle les pointeurs sur les entetes potentielles
             pICMPHeader       icmphdr        = (pICMPHeader)    ( (char*)recv_buf + (recv_buf->ihl * 4) );
           pIPHeader iphdr2                   = (pIPHeader)      ( (char*)icmphdr + sizeof(ICMPHeader) );
           pICMPHeader         icmphdr2   = (pICMPHeader)    ( (char*)iphdr2 + (iphdr2->ihl * 4) );


           //Teste les différents cas
           if (icmphdr->type == ICMP_TIME_EXPIRE)  //ICMP_TTL_EXPIRE = 11
           {
              printf("%s\n",inet_ntoa(*(in_addr*)&recv_buf->source_ip));
              break;
           }

           else if (icmphdr->type == ICMP_ECHO_REPLY) //ICMP_ECHO_REPLY = 0
```

```
            {
               if (icmphdr->seq != ttl_en_cours)
                  continue;

               printf("%s\n",inet_ntoa(*(in_addr*)&recv_buf->source_ip));
               ttl_en_cours = MAX_TTL;
               break;

            }

            else if (icmphdr->type == ICMP_DEST_UNREACH) //ICMP_DEST_UNREACH = 3
            {
               printf("x.x.x.x\n");
               break;
            }
            else
            {
               printf("?.?.?.? type %i\n",icmphdr->type);
               recu = true;
               Sleep(1000);
            }

         }//Fin du While de réception
      }//Fin du FOR de trace




cleanup:
   free(send_buf);
   free(recv_buf);
   closesocket(sd);
   WSACleanup();
   return 0;
}

u_short ip_checksum (u_short* buffer, int size)
{
   u_long sum = 0;
   while( size > 1 )  {sum += *buffer++;size -= 2;}

   if(size)         sum += * (u_char *) buffer;

   while (sum>>16)    sum = (sum & 0xffff) + (sum >> 16);

   return (u_short) ~sum;
}
```

IV.1 - Les entêtes dans la tête – TCP

TCP est un protocole plus complexe que IP. Mais dans notre cas, on n’a pas besoin de rentrer dans les détails. Notre usage permettra de se familiariser avec son entête sans être submergé.



```
+--------+--------+--------+--------+
|   PORT SOURCE   |PORT DESTINATION |
+--------+--------+--------+--------+
|        NUMERO DE SEQUENCE         |
+--------+--------+--------+--------+
|        NUMERO D'ACQUITEMENT       |
+--------+--------+--------+--------+
|HEADER L| FLAGS  |     FENETRE     |
+--------+--------+--------+--------+
|       CRC       |    PTR URGENT   |
+--------+--------+--------+--------+
|         OPTIONS ET PADDING
+--------+-----.......
```

Voici les différents flags avec leur valeur :

FIN	négocie une fin de connexion	1
SYN	négocie un début de connexion	2
RST	termine immédiatement	4
PSH	fonction push	8
ACK	accuse réception d’un paquet	16
URG	pointeur de données urgentes	32

De même, les numéros SEQ et ACK sont longuement détaillés. Je présente les notions de base. Lorsqu’on initialise une connexion, on choisit aléatoirement un numéro de séquence, qui sera le ISN (Initial Sequence Number). Le numéro de séquence augmente de paquet en paquet, non pas de un en un mais de la taille des données transmises. A un numéro SEQ ne correspond pas un paquet TCP mais un octet de donnée transmis. Lorsque l’ordinateur distant nous confirme réception d’un paquet ou d’une suite de paquet, il place dans son ACK le numéro SEQ qu’il s’attend à recevoir, c’est à dire 1 plus le SEQ du dernier octet de donnée bien reçu. Vouis suivez ?

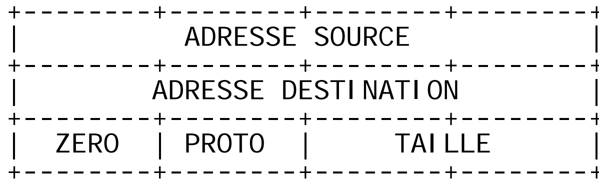
Le protocole définit également un tas de situations selon l’état de la connexion, géré par les flags. Ouverte ? Demande de fermeture non encore acceptée ? Pour le détail, la RFC est là. Voici sommairement comment se noue une connexion.

1. Le client se connecte au serveur, envoyant un paquet SYN et son ISNc (ISN client) dans SEQ.
2. Le serveur envoie un paquet SYN – ACK, disant qu’il a pris en compte la demande du client et en demande confirmation. Il envoie son ISNs (ISN serveur) dans SEQ, et ISNc+1 dans ACK.
3. Le client émet un paquet ACK pour confirmer la demande d’ouverture de connexion. Il met son ISNc+1 dans SEQ, et ISNs+1 dans son ACK.
4. La connexion est établie, le transfer de données peut débuter. Les numéros SEQ ne montent plus de 1 en 1, mais selon la taille du playload.

Là dessus vient se greffer le champ fenêtre. Mais comme pour notre exemple on va se contenter d’une utilisation simplifiée de TCP, je n’en parlerais pas.

On va mimer une connexion HTTP, un port souvent ouvert. En port source, mettons un port crédible dans le cas d’une requête web, genre 5000. Le port destination sera 80, forcément. Le numéro de séquence sera n’importe quoi aussi. Le numéro d’acquitement est nul car c’est nous qui initialisons la connexion. La taille de l’entête en mots de 32 bits est 5 : pas d’option. En Flags, on positionne le flag SYN. En fenêtre, peu importe. Pointeur urgent nul, et pas d’options.

Le calcul du CRC est un peu plus contraignant. Il faut faire précéder l’entête TCP d’une structure nomée Pseudo Header et qui contient les informanions importantes de l’entête IP que nous connaissons :


```
+--------+--------+--------+--------+
|          ADRESSE SOURCE           |
+--------+--------+--------+--------+
|        ADRESSE DESTINATION        |
+--------+--------+--------+--------+
|  ZERO  | PROTO  |      TAILLE     |
+--------+--------+--------+--------+
```

Tant que nous recevons des paquets ICMP TIME EXPIRED ou bien du time out, on n'a pas atteint notre destination. Si on reçoit un paquet TCP SYN ACK, on a touché au but et le port 80 est ouvert sur la cible. Il faut alors envoyer un paquet TCP RST pour clore la connexion. On peut aussi recevoir nous-même un paquet RST. Cela veut dire qu'on a atteint la cible mais que son port 80 est fermé. Ca n'est pas très important pour nous, sauf que du coup on n'est pas obligé d'émettre notre paquet RST. Pour alléger le code, je ne traite pas les flags reçus et émet systématiquement un RST. Dans ce paquet RST, le SEQ sera le ACK du paquet provenant de ma cible, et mon propre ACK sera nul.

```
genere_tcp( (char*)send_buf, local, distant,5000,80,TCP_RST,tcphdr->ack,0,ttl_en_cours);
```

Le programme est un peu plus complexe. Le socket est réglé pour accepter des entêtes IP utilisateur en émission. En écoute, il est réglé pour filtrer sur IP. Comme on va devoir traiter des paquets ICMP et TCP, on ne pouvait faire autrement. Ce socket est lié à l'adresse IP locale via un SockAddr. On a également déjà vu le petit bout de code pour déterminer l'IP la plus plausible. Le socket est ensuite débloqué en réception, afin de gérer à la main les timeouts.

Petite nouveauté, il faut faire passer le socket en mode Promiscuous. Dans ce mode, le socket communique tout ce qu'il entend. Même si ca n'arrive pas sur le port ou l'adresse du SockAddr auquel il est lié. Et même si le paquet n'est normalement pas retourné au programme car il ne porte pas de données mais sert juste, par exemple, dans la négociation d'une session. Ce mode est le propre des sniffers Ethernet qui s'en servent pour espionner toutes les trames qui transitent sur le bus. Dans notre cas, ce mode est nécessaire pour recevoir les paquets SYN ACK. Ca nous oblige à tester que l'IP Destination des paquets TCP qu'on reçoit sont bien à destination de notre IP locale, et pas à destionation d'un éventuel ordinateur présent sur le bus Ethernet.

```
unsigned int promiscuous = 1;
DWORD bytes;

WSAIoctl (parle, SIO_RCVALL, &promiscuous, sizeof (promiscuous), NULL, 0, &bytes, NULL, NULL) ;
```

J'ai regroupé le forgeage du paquet dans une fonction. Vous verrez comment j'utilise la place de l'entête IP pour écrire le pseudo header TCP afin de calculer le CRC TCP. Ensuite, je remplis l'entête IP qui écrase le pseudo header.

A propos de ce remplissage, gardez vos hton sous le coude ! Taille des paquets, numéros de ports, numéros de séquence et d'acquitement… Un oubli et votre entête sera invalide et donc inutilisable !

IV.2 – Utilité de la manipulation TCP

Le protocole TCP est le complément le plus courant de IP. Les champs critiques de TCP sont SEQ et ACK. On a vu que spoofer une adresse IP n'était pas très difficile. Cependant, il est plus difficile de spoofer un flux TCP valide. En effet, vu qu'on ne reçoit pas les réponses de la cible, son numéro SEQ est inconnu. Beaucoup d'études ont été faites sur la possibilité de prévoir les numéros SEQ. La méthode de *ISN Prediction* n'est ni impossible ni infaillible.

Il est également possible d'écouter une connexion en cours et, en répondant plus vite que l'hôte attendu et en se faisant passer pour lui, de transmettre un paquet TCP RST pour par exemple déconnecter un client IRC. Une fois réussie, un *TCP Hijacking* est la porte ouverte à bien d'autres malversations.

Accéder à TCP de manière brute permet également d'écouter et d'émettre sur un port local utilisé par un autre programme. Non seulement cela va à l'encontre du principe selon lequel un port pris par un programme lui est réservé, mais offre également la possibilité d'écouter et d 'émettre sans ouvrir de port et d'avoir le même genre de discrétion que dans l'utilisation d'ICMP.

IV.3 - Un traceroute en raw sockets TCP-IP

```
#include <winsock2.h>
#pragma comment (lib,"Ws2_32")
#include <stdio.h>
#include <ws2tcpip.h>
#include <winbase.h>


// Concernant ICMP-----------------------------------------------------------------------
  #define ICMP_ECHO_REPLY                        0
  #define ICMP_DEST_UNREACH                      3
    #define ICMP_DEST_UNREACH_NET_UNREACH                    0
    #define ICMP_DEST_UNREACH_HOST_UNREACH                   1
    #define ICMP_DEST_UNREACH_PROTOCOL_UNREACH               2
    #define ICMP_DEST_UNREACH_PORT_UNREACH                   3
    #define ICMP_DEST_UNREACH_FRAG_NEEDED_DF                 4
    #define ICMP_DEST_UNREACH_SOURCE_ROUTE_FAILED            5
  #define ICMP_SOURCE_QUENCH                     4
  #define ICMP_REDIRECT                          5
    #define ICMP_REDIRECT_NETWORK                            0
    #define ICMP_REDIRECT_HOST                               1
    #define ICMP_REDIRECT_TOS_NETWORK                        2
    #define ICMP_REDIRECT_TOS_HOST                           3
  #define ICMP_ECHO_REQUEST                      8
  #define ICMP_TIME_EXPIRE                       11
    #define ICMP_TIME_EXPIRE_TTL                             0
    #define ICMP_TIME_EXPIRE_FRAG_REASSEMBLY_EXPIRE          1
  #define ICMP_PARAM_PROBLEM                     12
    #define ICMP_PARAM_PROBLEM_POINTER_SET                   0
  #define ICMP_TIMESTAMP_REQUEST                 13
  #define ICMP_TIMESTAMP_REPLY                   14
  #define ICMP_INFO_REQUEST                      15
  #define ICMP_INFO_REPLY                        16

  // ICMP header
  typedef struct
  {
    BYTE type;
    BYTE code;
    USHORT checksum;
    USHORT id;
    USHORT seq;
  } ICMPHeader, *pICMPHeader;


// Concernant IP-------------------------------------------------------------------------
  #define IP_PROTO_ICMP      1
  #define IP_PROTO_TCP       6
  #define IP_PROTO_UDP       17

  // The IP header
  typedef struct
  {
    BYTE ihl:4;
    BYTE version:4;
    BYTE tos;
    USHORT total_len;
    USHORT ident;
    USHORT flags;
    BYTE ttl;
    BYTE protocol;
    USHORT checksum;
    ULONG source_ip;
    ULONG dest_ip;
  } IPHeader, *pIPHeader;


// Concernant TCP------------------------------------------------------------------------
  #define TCP_FIN            1
  #define TCP_SYN            2
  #define TCP_RST            4
```

```
   #define TCP_PSH              8
   #define TCP_ACK              16
   #define TCP_URG              32

   typedef struct
   {
      USHORT source_port;
      USHORT dest_port;
      ULONG seq;
      ULONG ack;
      BYTE reserved:4;
      BYTE data:4;
      BYTE flags;
      USHORT window;
      USHORT checksum;
      USHORT urg_pointer;
   } TCPHeader, *pTCPHeader;

   typedef struct
   {
      ULONG source_ip;
      ULONG dest_ip;
      BYTE zero;
      BYTE protocol;
      USHORT taille;
   } PseudoHeader, *pPseudoHeader;

//-----------------------------------------------------------------------------------------


   #define TCP_SIZE 20
   #define MAX_TTL 30
   #define MAX_ECOUTE_SIZE (1024 + sizeof(IPHeader))
   #define RETRY_LOOP 20
   #define RETRY_WAIT 300

   USHORT ip_checksum(USHORT* buffer, int size);
   void genere_tcp (char* buffer, ULONG source_ip, ULONG dest_ip, USHORT source_port, USHORT dest_port, BYTE flags, ULONG syn, ULONG
ack, CHAR ttl);


   int global_size;


int main(int argc, char* argv[])
{
   // VARIABLES : les buffers
      pIPHeader send_buf          = NULL;
      pIPHeader send_ip           = NULL;
      pTCPHeader send_tcp         = NULL;
      pIPHeader recv_buf          = NULL;
      pPseudoHeader send_pseudo = NULL;

   // VARIABLES : pour la destination et la source
      sockaddr_in sa_dest, sa_local;
      int taille_sa_local = sizeof(sockaddr);
      hostent* he_resolu;
      ULONG local,distant;

   // VARIABLES : pour le ping
      SOCKET parle, ecoute;

   // COMPTEUR pour le trace
      int ttl_en_cours,attente;
      int compte_result = 0;


   // ETAPE 1 : GERER LA LIGNE DE COMMANDE
      // Message d'erreur en cas de manque de paramètres
         if (argc < 1)
            {printf("usage: rawping <host> \n");exit (1);}

         global_size = sizeof(IPHeader) + TCP_SIZE;
```

```
// ETAPE 2 : INITIALISATION DES RESSOURCES RESEAU
  // Lance WinSocks
    WSAData wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0)
      {printf("Impossible de démarrer WinSock. Version peut-être inférieure à 2.1\n");return 1;}

  // Alloue les buffers. Le source est indexé convenablement
    send_buf = (pIPHeader) malloc(global_size);
    if (send_buf == NULL)
      {printf("Impossible d'allouer le buffer d'emission\n");goto cleanup;}
    memset(send_buf,0,global_size);
    send_ip =  (pIPHeader)                    send_buf;
    send_tcp =(pTCPHeader)  ( (char*)send_buf+ sizeof(IPHeader) );
    send_pseudo = (pPseudoHeader) ( (char*)send_tcp-sizeof(PseudoHeader) );

    recv_buf = (pIPHeader)malloc(MAX_ECOUTE_SIZE);
    if (recv_buf == NULL)
      {printf("Impossible d'allouer le buffer de reception\n");goto cleanup;}


  //Calcule les adresses locales et distantes
    //La distante depuis argv[1]
    he_resolu = gethostbyname(argv[1]);
    distant = *(ULONG*)he_resolu->h_addr_list[0];

    //Calcul de la meilleure IP locale
    {
      //Chope mon ip : résolution de mon nom
      char hostname[1024];
      gethostname(hostname,1024);
      he_resolu = gethostbyname(hostname);

      //Recherche de l'ip la plus basse
      int laquelle_est_bonne = 0;
      unsigned char valeur = 255;
      int i=0;
      while (he_resolu->h_addr_list[i] != 0)
      {
        if (*(unsigned char*)he_resolu->h_addr_list[i] < valeur)
        {
          laquelle_est_bonne = i;
          valeur = *(unsigned char*)he_resolu->h_addr_list[i];
        }
        i++;
      }

      local = *(ULONG*)he_resolu->h_addr_list[laquelle_est_bonne];
    }


  //Affiche l'état actuel
    printf("résoultion de %s en %s\n",argv[1], inet_ntoa( *(in_addr*) &distant) );
    printf("mon ip est : %s\n\n",inet_ntoa( *(in_addr*) &local) );


  // Créé le RAW SOCKET TCP, le règle en IP HEADER inclus et non bloquant
    {
      BOOL hdrincl = 1;
      unsigned long son_masque_ioctlsocket = 1;
      #define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
      unsigned int promiscuous = 1;
      DWORD bytes;

      //Créé le socket
      parle = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
      if (parle == INVALID_SOCKET)
        {printf("Erreur de création du Raw Socket : %s\n",WSAGetLastError() ); goto cleanup;}

      //Créé le sockaddr distant pour l'émission
      memset(&sa_dest,0,sizeof(sockaddr));
      sa_dest.sin_family = AF_INET;
      sa_dest.sin_port = 0;
      *(ULONG*)&sa_dest.sin_addr = distant;
```

```
        //Créé le sockaddr local pour l'écoute
        memset(&sa_local,0,sizeof(sockaddr));
        *(ULONG*)&sa_local.sin_addr = local;
        sa_local.sin_family = AF_INET;
        sa_local.sin_port = htons(5000);

        //Lie le socket
        if (bind (parle, (struct sockaddr *) &sa_local, taille_sa_local) == SOCKET_ERROR)
           {printf ("Impossible de lier le socket d'ecoute\n");goto cleanup;}

        //Le débloque
        if (ioctlsocket (parle, FIONBIO, &son_masque_ioctlsocket)!=0)
           {printf("Impossible de débloquer le socket : %s\n",WSAGetLastError() );goto cleanup;}

        //Règle l'option d'entête IP
        if (setsockopt(parle, IPPROTO_IP, IP_HDRINCL, (const char*)&hdrincl, sizeof(hdrincl)) == SOCKET_ERROR)
           {printf("Impossible de régler le header ip inclus : %s\n",WSAGetLastError() );goto cleanup;}

        //Le passe en promiscuous
        if (WSAIoctl (parle, SIO_RCVALL, &promiscuous, sizeof (promiscuous), NULL, 0, &bytes, NULL, NULL) == SOCKET_ERROR)
           {printf("Promiscuous impossible\n"); goto cleanup;}


      }

/////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
//                    //
// LA BOUCLE DE TRACE //
//                    //
////////////////////////
    for (ttl_en_cours = 19; ttl_en_cours < MAX_TTL ;ttl_en_cours++)
    {
      int taille;

  // ETAPE 3 : REGLE LES ENTETES
      genere_tcp( (char*)send_buf, local, distant,5000,80,TCP_SYN,12000,0,ttl_en_cours);
      pTCPHeader bibi = (pTCPHeader) ( (char*)send_buf + sizeof(TCPHeader));

  // ETAPE 4 : ACTION !!
    // Envoi du ping, avec le ttl voulu
      Sleep(RETRY_WAIT);
      taille = sendto(parle, (char*) send_buf, global_size, 0, (sockaddr*)&sa_dest, sizeof(sa_dest));
      if ( (taille == SOCKET_ERROR) | (taille != global_size) )
        {printf("Erreur d'emission %i\n", WSAGetLastError());goto cleanup;}

    // Boucle de réception
      bool recu = false;
      while (!recu)
      {

        //Ecoute
        memset(recv_buf,0,MAX_ECOUTE_SIZE);
        taille = 0;

        for (attente=0; attente<RETRY_LOOP; attente++)
        {
          taille = recvfrom(parle, (char*)recv_buf, MAX_ECOUTE_SIZE, 0,(struct sockaddr *) &sa_local,&taille_sa_local);

          //Si erreur
          if (taille == SOCKET_ERROR)
          {
            //Et que besoin de plus de temps
            if (WSAGetLastError()==WSAEWOULDBLOCK) //WSAEWOULDBLOCK = 10035
              //on attend
              Sleep(RETRY_WAIT);
            else
            {
              //si l'erreur est grave
              printf("Erreur de réception %i\n",WSAGetLastError());
              break;
            }
          }//fin du si erreur

          //Si pas d'erreur et taille non nulle on a assez entendu
```

```
            else if (taille != 0)
                break;
        }

        //Si l'écoute a causé une erreur, on affiche X et on passe au hop suivant
        if (taille <= 0)
        {
            printf("%i\tx.x.x.x - %i\n",++compte_result, WSAGetLastError());
            break;
        }

        //Si je ne suis pas sur un ICMP. Est-ce mon SYN ACK ?
        if (recv_buf->protocol == 6)
        {
            pTCPHeader tcphdr = (pTCPHeader) ( (char*)recv_buf + (recv_buf->ihl * 4) );

            if (  recv_buf->source_ip == send_ip->dest_ip &&            //ip origine = ma cible
                  tcphdr->source_port == send_tcp->dest_port     &&     //port origine = ma cible
                  tcphdr->dest_port == send_tcp->source_port )   //port dest = mon port
            {
                //Si oui, tue la connexion
                genere_tcp( (char*)send_buf, local, distant,5000,80,TCP_RST,tcphdr->ack,0,ttl_en_cours);
                printf("%i\t%s\n",++compte_result,inet_ntoa(*(in_addr*)&recv_buf->source_ip));
                ttl_en_cours = MAX_TTL;
            }
            break;
        }

        //Règle les pointeurs sur les entetes potentielles
          pICMPHeadericmphdr  = (pICMPHeader)( (char*)recv_buf + (recv_buf->ihl * 4) );
        pTCPHeader  tcphdr= (pTCPHeader)( (char*)recv_buf + (recv_buf->ihl * 4) );
        pIPHeader iphdr2= (pIPHeader)       ( (char*)icmphdr + sizeof(ICMPHeader) );
        pICMPHeadericmphdr2 = (pICMPHeader)( (char*)iphdr2 + (iphdr2->ihl * 4) );


        //Teste les différents cas
        if (icmphdr->type == ICMP_TIME_EXPIRE)  //ICMP_TTL_EXPIRE = 11
        {
            printf("%i\t%s\n",++compte_result,inet_ntoa(*(in_addr*)&recv_buf->source_ip));
            break;
        }

        else if (icmphdr->type == ICMP_DEST_UNREACH) //ICMP_DEST_UNREACH = 3
        {
            printf("%i\tx.x.x.x\n",++compte_result);
            break;
        }
        else
        {
            printf("%i\t?.?.?.? type %i\n",++compte_result,icmphdr->type);
            recu = true;
            Sleep(1000);
        }

      }//Fin du While de réception


    }//Fin du FOR de trace



    printf("\n\n");

cleanup:
    free(send_buf);
    free(recv_buf);
    closesocket(parle);
    closesocket(ecoute);
    WSACleanup();
    return 0;
}



u_short ip_checksum (u_short* buffer, int size)
```

```
{
   u_long sum = 0;
   while( size > 1 )      {sum += *buffer++;size -= 2;}

   if(size)         sum += * (u_char *) buffer;

   while (sum>>16)                    sum = (sum & 0xffff) + (sum >> 16);

   return (u_short) ~sum;
 }


void genere_tcp (char* buffer, ULONG source_ip, ULONG dest_ip, USHORT source_port, USHORT dest_port, BYTE flags, ULONG syn,
ULONG ack, CHAR ttl)
 {
        pIPHeader send_ip =         (pIPHeader)               buffer;
        pTCPHeader send_tcp =    (pTCPHeader)           ( (char*)send_ip+ sizeof(IPHeader) );
        pPseudoHeader send_pseudo = (pPseudoHeader) ( (char*)send_tcp-sizeof(PseudoHeader) );

        send_tcp->source_port = htons(source_port);
        send_tcp->dest_port = htons(dest_port);
        send_tcp->seq = syn;
        send_tcp->ack = ack;
        send_tcp->data = 5;
        send_tcp->reserved = 0;
        send_tcp->flags = flags;
        send_tcp->window = 65535;
        send_tcp->checksum = 0;
        send_tcp->urg_pointer = 0;

        send_pseudo->source_ip = source_ip;
        send_pseudo->dest_ip = dest_ip;
        send_pseudo->zero = 0;
        send_pseudo->protocol = IP_PROTO_TCP;
        send_pseudo->taille = htons(TCP_SIZE);

        // Calcule le CRC
        send_tcp->checksum = ip_checksum((USHORT*)send_pseudo, TCP_SIZE+ sizeof(PseudoHeader));

      //S'occupe de l'entête IP
        send_ip->version = 4;
        send_ip->ihl = 5;
        send_ip->tos = 0;
        send_ip->total_len = htons(global_size);
        send_ip->ident = 0;
        send_ip->flags = 0;
        send_ip->ttl = ttl;
        send_ip->protocol = IP_PROTO_TCP;
        send_ip->checksum = 0;
        send_ip->source_ip = source_ip;
        send_ip->dest_ip =   dest_ip;
        send_ip->checksum = ip_checksum((USHORT*)send_ip, sizeof(IPHeader));
}
```

VI – Conclusion

Le programme est loin d'être parfait. Beaucoup de vérifications ne sont pas faites. Si vous mettez en hôte www.gaagle.com, vous n'aurez mas de message d'erreur pour nom impossible à résoudre mais un plantage pur et simple. Programmation offensive, l'utilisateur n'a qu'à pas être con ! De même, en environnement très dense en connexions, le tri insuffisant des paquets reçus fait que le programme donne des résultats plutôt… imparfaits. Ca serait mieux sans le mode promiscuous, deja… On pourrait ainsi lancer une rafale de requêtes et attendre les réponses également en vrac au lieu de traiter au cas par cas. Du coup, on "casse" les longs timeouts dû au fonctionnement connecté, demande / réponse, de TCP.

Désormais, vous avez à disposition quelques brèves explications et des codes sources fonctionnels qui vous ont présenté le forgeage des protocoles IP, TCP, ICMP, ainsi que l'ouverture d'un socket pour le sniff.

Pas mal en 35 pages, non ?