LES RAW-SOCKETS EN C SOUS LINUX

Introduction

Dans cet article je vais tenter (je dis bien tenter) de vous expliquer comment utiliser les RAW Sockets en C sous UNIX. Je tiens a preciser que cet article a ete ecrit a partir de la traduction de "Documentation about native raw socket programming" ecrit par Nitr0gen de Exile 2000 (www.exile2k.org). J'espere que son auteur ne verra pas d'inconvenient a une telle utilisation de sa documentation, dans ce cas, s'il lit ces mots et qu'il est en desaccord, qu'il me mail. Fermons maintenant cette parenthese pour revenir a nos moutons. Cet article est le premier d'une suite qui vous expliquera la programmation avec les RAW_SOCK. Dans cet article vous apprendrez a faire vous-memes un header IP. J'attends de vous d'avoir des connaissances en C et en TCP/IP, car sinon vous ne comprendrez rien... Sur ce, bonne lecture :)

1) Definition

Qu'est-ce qu'un socket RAW certains pourront se demander... Un RAW socket est un socket (sans blagues ? ;) dans lequel les champs des en-tetes des paquets envoyes sont remplis a la main. On accede donc a un niveau de programmation reseau assez bas puisqu'on atteins directement les couches de TCP/IP. Cela va nous permettre de forger des paquets comme bon nous semble et ainsi de nous adonner a des occupations telles que le spoofing (RIP,ICMP,IP...) le hijacking, le portmap 'furtif' et bien d'autres choses encore!

2) Comment programmer un RAW_Socket?

En soit, programmer avec des RAW n'est pas tres complique, il suffit de savoir faire (comme toutes choses vous me direz). Il suffit de remplir les structures concernees et de faire un appel a la fonction socket() remplie comme suit (soit dit en passant, si vous n'y connaissez rien en programmation reseau, allez lire l'article sur ce sujet dans ce mag, c'est indispensable!) :

```
socket(DOMAIN,TYPE,PROTO)
```

Dans DOMAIN on met AF_INET pour l'IPv4 ou AF_INET6 pour l'IPv6.
Dans TYPE, vu qu'on fait des sockets RAW, on met SOCK_RAW.
Dans PROTO, et bien ca depend du protocole que l'on veut utiliser. Allez voir votre /etc/protocols et trouvez le nombre correspondant : ce sera la valeur a mettre a la place de PROTO.

Ok, maintenant on va remplir nos en-tetes, puis on va envoyer nos paquets tout simplement avec "send". Voyons comment remplir l'en-tete IP...

3) Comment remplir l'en-tete IP

Voyons tout d'abord la structure correspondant a un tel en-tete :

```
struct iphdr {
#if __BYTE_ORDER == __LITTLE_ENDIAN
        unsigned int ihl:4;
        unsigned int version:4;
#elseif __BYTE_ORDER == __BIG_ENDIAN
        unsigned int version:4;
        unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
        u_int8_t tos;          /* correspond au TOS */
        u_int16_t tot_len;   /* correspond a la taille totale */
        u_int16_t id;          /* correspond a l'identification */
        u_int16_t frag_off;  /* correspond au fragment offset */
        u_int8_t ttl;          /* correspond au TTL */
        u_int8_t protocol;   /* correspond au protocole de transport */
        u_int16_t check;    /* correspond a la somme de controle */
        u_int32_t saddr;     /* correspond a l'IP source */
        u_int32_t daddr;    /* correspond a l'IP destination */
        /* the options start here. */
};
```

(Nota : les commentaire francais ne sont pas dans le .h a la base, c'est moi qui les ai rajoute pour cet article).
Bon, voila notre structure a remplir. Maintenant, interessons nous au shema de l'en-tete IP d'un paquet...

<table>
<tr><td>0</td><td></td><td>15-16</td><td colspan="2">31</td></tr>
<tr><td>version
(4)</td><td>IHL
(4)</td><td>TOS
(8)</td><td colspan="2">Taille totale
(16)</td></tr>
<tr><td colspan="3">Identification
(16)</td><td>Flags
(3)</td><td>Frag Offset
(13)</td></tr>
<tr><td colspan="2">TTL
(8)</td><td>Protocole
(8)</td><td colspan="2">Somme de controle
(16)</td></tr>
<tr><td colspan="5">Adresse IP source
(32)</td></tr>
<tr><td colspan="5">Adresse IP destination
(32)</td></tr>
<tr><td colspan="5">Options
(si il en a)</td></tr>
<tr><td colspan="5">Donnees</td></tr>
</table>

Ok, les chiffres entre parentheses representent la longueur, en bits, des champs. La taille de l'en-tete IP options et donnees exceptees est de 20 octets. Voyons a quoi correspondent les differents champs :

*Version(4bits) : il s'agit de la version d'IP utilisee. Actuellement nous en sommes a IPv4 (donc version 4), IPv6 va bientot etre officialise.
*IHL(4bits) : l'IHL (Internet Header Lenght) est la taille, en words de 32bits (donc en DWord, ceux qui ignorent a quoi cela correspond allez apprendre l'ASM), de l'en-tete IP du paquet. Si aucune option n'est utilisee, la valeur de l'IHL devrait etre 5.
*TOS(8bits) : TOS (Type Of Service) est utilise pour specifier de quels services on veut profiter.

----------{EXPRESS NOTE}----------

Nous avons 4 choix pour remplir le champs TOS. Voici leur nom et leur
valeur en hexadecimal :

NOM	Valeur
1- Minimize delay	0x10
2- Maximize throughtput	0x08
3- Maximize reliability	0x04
4- Minimize monatary cost	0x02

1: Ceci est utilise par les applications qui envoient de petits paquets de donnees et qui ont besoin d'une reponse rapide.
2: Ceci, a contrario du premier, est utilise par les applications qui envoient beaucoup de donnees.
3: prime sur la qualité de la connexion
4: prend le chemin qui occasionnera le moins de cout monétaire

----------{END OF EXPRESS NOTE}----------

* Taille totale(8bits) : ce champs correspond a la taille du datagramme. Par exemple nous avons un header (= en-tete, mais header c'est plus court ;) IP et un header TCP (flag syn) sans donnee. Ces deux headers ayant chacun une taille de 20, nous mettrons dans le champs de la taille totale du header IP 40 (20+20).
* Identification(16bits) : l'identification est utilisee pour identifier les fragments. Quand les datagrammes ne

sont pas
fragmentes, ce champs est inutile. Generalement, l'identification est incrementee de datagrammes en datagrammes. Chaque fragment a la meme identification que le premier datagramme.
* Flags(3bits) : Attention, il ne s'agit pas des flags des headers TCP (SYN, ACK,FIN,RST,...), ne confondez pas. Ce champs est utilise pour la fragmentation.

----------{EXPRESS NOTE}----------

Nous avons 4 flags differents. Les voici :

Nom	Valeur(hexa)
No flags	0x00
More fragment	0x01
Don't fragment	0x02
More and don't fragment	0x03

More fragment indique qu'il y a d'autres fragments apres ce paquet, don't fragment indique de ne pas fragmenter ce paquet et More and don't fragment est la combinaison des deux. Il est evident que lorsque vous
fragmentez un datagramme, le dernier paquet n'a pas le flag More Fragment!

----------{END OF EXPRESS NOTE}----------

* Fragment Offset(13bits) : Ceci est l'offset a l'interieur du paquet. Le datagramme aura un offset de 0. Ce champs est calcule sur 64bits. Le dernier offset sera egal a tot_len.
* TTL(8bits) : Le champs TTL (time to live) est utilise pour specifier combien de sauts (etudiez le routage des reseaux et RIP si vous ne savez pas a quoi ceci correspond) que le datagramme va pouvoir effectuer avant d'etre balance aux oubliettes. A chaque saut, le TTL est decremente de 1. Lorsque le TTL atteins 0, le paquet "meurt". Ceci est utilise afin de ne pas avoir de paquets lances dans des boucles infinies sur un reseau (si on pouvait faire de telles boucles, je ne vous explique meme pas le flood qu'on pourrait balancer sur le reseau!!!!).
* Protocol(8bits) : Ce champs est utilise pour indiquer le protocole de la couche transport utilise. Pour connaitre ces protocoles, allez voir dans '/usr/include/linux/in.h'. Sinon, voici les trois principaux :
IPPROTO_TCP 0x06
IPPROTO_UDP 0x11
IPPROTO_ICMP 0x01

* Somme de controle(16bits) : La somme de controle est utilisee pour verifier l'integrite des datagrammes et donc eviter toute modification exterieure de ces derniers. Voyez le paragraphe pour la calculer.
* Adresse IP source(32bits) : IP source du datagramme.
* Adresse IP destination(32bits) : IP destination du datagramme.
* Options (taille variable) : je ne m'attarderai pas sur ce point.

4) La fragmentation

Il y a fragmentation lorsque le MTU (Maximum Transfert Unit) est inferieur a la taille totale du datagramme. Dans ce cas la, on opere une fragmentation : le datagramme est divise en plusieurs pieces plus petites que l'on envoie. A l'arrivee, les pieces sont reassemblees pour reformer le datagramme. Mais, pour que la machine de destination puisse reformer le
datagramme convenablement il faut que :

* Le flag 'More Fragment' soit mis a tous les fragments sauf le dernier
* Le Fragment Offset pour le premier paquet doit etre a 0
* L'identification doit etre la meme pour tous les fragments, pour savoir a quel datagramme appartient quel fragment
* Si le header IP d'un fragment est modifie, la somme de controle doit etre recalculee
* La taille totale des premiers fragments a pour valeur le MTU

5) Le calcul de la somme de controle

Calculer la valeur de la somme de controle n'est pas complique, il suffit d'utiliser cette fonction :

unsigned short in_cksum(unsigned short *addr, int len);

- unsigned short *addr est un pointeur sur le header IP (donc sur la structure)
- int len est la taille du header IP

6) Exemple de code

Voici un exemple de code tire de l'article de Nitr0gen. Meme si j'ai supprime le commentaire ou le copyright figurait, ce dernier s'applique toujours. J'ai traduit les commentaires. La fonction buildip_nf() forge un header IP sans fragmentation, la fonction buildip_f() forge un header IP avec fragmentation : le datagramme est separe en deux fragments et le MTU est de 280 octets...

```
/* debut du code */

void buildip_nf() {         /* fonction forgeant un header IP */

        struct iphdr *ip;
                        /* un petit pas pour l'homme, un grand pas pour l'humanite */

        ip = (struct iphdr *) malloc(sizeof(struct iphdr));
                    /* allocation dynamique de memoire */

        ip->ihl = 5;                /* IHL en octets */
        ip->version = 4;            /* version du protocole IP */
        ip->tos = 0;               /* tos est une implementation experimentale de IP */
        ip->tot_len = sizeof(struct iphdr) + 452 /* taille totale du paquet */

        ip->id = htons (getuid());
                    /* identification du paquet, inutile pour nous */

        ip->ttl = 255;                          /* le paquet peut effectuer 255 sauts */
        ip->protocol = IPPROTO_TCP;             /* si nous utilisons TCP en transport */
        ip->saddr = inet_addr("127.0.0.1");     /* ip source */
        ip->daddr = inet_addr("127.0.0.1");     /* ip destination */

        ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));
         /* somme de controle */
}


void buildip_f() {
        struct iphdr *ipf;

        ipf = (struct iphdr *) malloc(sizeof(struct iphdr));

/**** PREMIER FRAGMENT ****/
        ipf->ihl          = 5;
        ipf->version   = 4;
        ipf->tos        = 0;
        ipf->tot_len   = sizeof(struct iphdr)+256 /*taille du premier fragment*/
        ipf->id          = htons(1); /* pour identifier nos fragments */
        ipf->ttl         = 255;
        ipf->protocol = IPPROTO_TCP;
        ipf->saddr     = inet_addr("127.0.0.1");
        ipf->daddr     = inet_addr("127.0.0.1");
        ipf->frag_off  = htons(0x2000); /* Fragment 0 et More Fragment */
        ipf->check   = in_cksum((unsigned short *)ipf,sizeof(struct iphdr)+256);

                    /* ON ENVOIE LE PREMIER FRAGMENT ICI */

/**** DEUXIEME FRAGMENT ****/
        ipf->tot_len = sizeof(struct iphdr) + 196 /* update de la taille */
        ipf->frag_off = htons(32); /* offset du fragment, pas de MF */
        ipf->check = incksum((unsigned short *)ipf,sizeof(struct iphdr)+196);
```

```
/* on est force de recalculer la somme de controle */
/* vu qu'on a modifie l'en-tete du paquet */

/* ON ENVOIE LE SECOND FRAGMENT ICI */}
```

7) Conclusion

Voila, c'est tout pour l'instant. Dans la prochaine partie de cet article, qui paraitra dans Counter Strike #4, vous apprendrez a forger des headers TCP et UDP. En attendant, amusez-vous avec le peu que je vous ai donné...