IMPLEMENTAZIONE IN C DI UN KEYLOGGER
IN AMBIENTE UNIX/LINUX

UNIVERSITÀ DEGLI STUDI DI NAPOLI
“PARTHENOPE”

Implementazione a cura di

FIORENTINO MICHELE

GUIDA ALL'USO

La cartella "codice" contiene i seguenti file:

- CrackedSoftware.c
- kserver.c
- keylogger.h
- keyboard.h
- berkeleySockets.h

CrackedSoftware e kserver avranno in comune tutti e 3 gli header file, e sono divisi in due cartelle: "Client" e "Server", che dovrebbero simulare lo spazio di lavoro di due computer diversi.

L'utente che compila la parte client del codice malevole utilizza un Makefile, anche per nascondere ulteriormente i file contenenti il codice del keylogger.

```
make
```

Il server potrà facilmente compilare la parte server del codice come:

```
gcc kserver.c -o kserver.out
```

CRACKEDSOFTWARE.C

È un fantoccio. Rappresenta un qualsiasi codice nel quale sia stato inserito il virus. Banalmente, al proprio interno verrà inclusa la libreria "keylogger.h" e verrà richiamata la funzione "activate_keylogger()" con il quale viene attivato il keylogger.

KSERVER.C

Si tratta del programma che si occupa di gestire le richieste che provengono dai client, ovvero da coloro che sono infettati con il keylogger. È stato implementato come un server multithreaded: ogni qualvolta viene accettata una nuova richiesta, viene generato un nuovo thread che avrà il compito di occuparsene.
Il thread si occuperà di recuperare il nome e i dati del client, ovvero i tasti battuti.
Le informazioni verranno salvate in un file "*cod_utente*.txt".

KEYLOGGER.H

È il cuore del keylogger. Avvia una connessione verso il server (per motivi pratici, l'IP posto è "127.0.0.1", ma va bene qualsiasi IP). Il keylogger si occuperà di fornire al server il proprio nome e tutti i tasti che vengono battuti dall'utente.

In Linux puntiamo a leggere il buffer della tastiera, e questo buffer si trova in /dev/input/by-path. Possiamo notare che ci sono diversi file, ma l'unico che ci interessa è il *-kbd, in quanto su tratta di un link dinamico al buffer della tastiera vero e proprio, dunque leggeremo da lì.

Per leggere utilizzeremo le normali funzioni per la manipolazione di file "open()", "read()" e "close()" presenti nella libreria fcntl.h.

Ogni qualvolta viene premuto un tasto, viene generato un evento input. Noi siamo interessati ad intercettarli.

Faremo utilizzo di una struct input_event.

Tale struct è così composta:

```
struct input_event {
struct timeval time;
__u16 type;
__u16 code;
__s32 value;
};
```

dove:

- time: rappresenta la marca temporale. Ritorna il tempo al quale l'evento si è verificato. È in secondi o millisecondi;
- type: indica il tipo di evento. Ad esempio EV_KEY è utilizzato per descrivere i cambiamenti di stato della tastiera (o di dispositivi simili), mentrew EV_REL è utilizzato per descrivere i cambiamento di stato di un mouse.
- code: indica il codice del tasto premuto. Ad esempio KEY_A rappresenta il tasto A.
- value: indica il valore che ha l'evento. Può essere un cambio relativo per EV_REL, o per EV_KEY può essere 0 per il rilasci o 1 per la pressione (ce ne sono altri).

L'idea è creare polling, aspettando la pressione di un tasto. Se Il tipo di EV riguarda un cambiamento di stato della tastiera (EV_KEY, ergo è stato premuto/rilasciato un tasto) e il valore è 1 o 0 (dunque è stato premuto o è stato rilasciato), andiamo a salvare il valore di questo tasto all'interno di una stringa.

Notare che prima di spedire il valore dovremo tradurlo, in quanto "code" restituisce un intero (ad esempio, "a" restituisce il codice 30). Quest'operazione verrà effetttuata dalla funzione "getCodeMeaning(ev.code, ev.value)", che restituisce un char.

KEYBOARD.H

il file keyboard.h contiene funzioni che permettono di convertire un valore di tipo “ev.code” (un intero) in un carattere leggibile.
In particolar modo, verrà richiamata la funzione getCodeMeaning(int code, int value), al quale passiamo il codice (numerico) del carattere e l’azione compiuta (PRESSIONE/RILASCIO).

VIENE PRESO IN CONSIDERAZIONE IL **LAYOUT US**, e non quello IT.

La conversione dei caratteri avviene attraverso una corrispondenza diretta fra “ev.code” e il carattere che rappresenta. Possiamo verificare tali corrispondenze analizzando il file al seguente link Github: https://github.com/torvalds/linux/blob/master/include/uapi/linux/input-event-codes.h

Per motivi pratici non ho implementato ogni singola corrispondenza, ma solo quelle che ho ritenuto utili (i caratteri dell’alfabeto più qualche simbolo speciale).
Tali corrispondenze sono definite all’interno di un array “dict” il quale contiene, per ogni ev.code, il suo carattere corrispondente.
Determinati caratteri sono ottenuti da una combinazione di tasti (ad esempio, ‘!’ è ottenuto da LEFT SHIFT + ‘1’). I tasti che sono pensati per essere combinati con altri tasti (es. LEFT SHIFT) li consideriamo “tasti speciali”.
Quando teniamo premuto un “tasto speciale”, questo rimane attivo finché non viene rilasciato.
Attraverso una variabile “status” teniamo traccia di tutti i tasti speciali attualmente attivi.

Per una questione di efficienza, la variabile status non è altro che un char che rappresenta un insieme di valori booleani (es. 0000 1001), dove ogni valore booleano rappresenta una flag. Ad ogni tasto speciale verrà assegnato un VALORE potenza di 2, proprio perché occupa una specifica posizione in status.

La logica è che se voglio rappresentare uno stato dove sia CTRL LEFT che ALT LEFT sono premuti contemporaneamente, allora la variabile status avrà la flag alzata sia per CTRL LEFT che per ALT LEFT:

status = 1001 (data da CTRL LEFT + ALT LEFT, ovvero 0001 + 1000).

Il valore associato ad ogni tasto verrà restituito da un’apposita funzione getNewStatus(unsigned char ch). Solo nel caso di tasti speciali verrà restituito un valore diverso da 0.

Ogni qualvolta un tasto speciale viene premuto questo altera la variaile status, alzando il corrispettivo flag. Se invece un tasto speciale viene lasciato, la corrispettiva flag nella variabile status viene abbassata.

L’unico tasto speciale che fa eccezione è il CAPS LOCK. Infatti questo si attiverà/disattiverà ciclicamente ogni volta che lo premeremo. L’effetto è che tutti i caratteri dell’alfabeto passeranno dall’essere in minuscolo a maiuscolo.

Se CAPS LOCK viene usato in combinazione di LEFT SHIFT, il suo effetto viene annullato (le minuscole rimangono minuscole).

Quando utilizziamo una combinazione di LEFT SHIFT + un qualsiasi altro tasto, verifichiamo qual è il carattere dato da tale combinazione scorrendo all’interno di uno switch.

La funzione “getShiftChar(uncisgned char ch)” si occuperà di restituire il tasto premuto in corrispondenza di uno SHIFT.

BERKELETSOCKETS.H

Si tratta di una libreria che contiene le funzioni wrapper per le system call utilizzate dalle socket di Berkeley. Ciò permette agli altri file del progetto di richiamare le suddette system call senza preoccuparsi della gestione di eventuali errori connessi al loro utilizzo.

Socket(), Connect(), Bind(), Listen(), Accept() e Close() sono autoesplicative.

getInitAddr() si tratta di inizializzare l'indirizzo specificato, richiamare la funzione inet_pton, e di gestirne gli eventuali errori.

La funzione **fullWrite()** si occupa di scrivere esattamente *count* byte, iterando opportunamente le scritture. Se si verifica un'interruzione, il ciclo viene ripetuto (a meno che non si verifichi un errore irreversibile).
Gli argomenti sono gli stessi della write: il fd della socket, il buffer e la sua dimensione.

Nel caso la write restituisse un valore < 0, possono essersi presentate una di due situazioni:

- è stata generata un'interrupt da una chiamata di sistema (errno == EINTR), in tal caso ripeteremo banalmente il ciclo;
- altrimenti si è presentato un qualche altro tipo di errore, che ritorneremo con una exit().

Alla fine restituisco nleft, che è 0.

La funzione **fullRead()** si occupa di leggere esattamente *count* byte, iterando opportunamente le letture. Se si verifica un'interruzione, il ciclo viene ripetuto (a meno che non si verifichi un errore irreversibile).

Gli argomenti sono gli stessi della read: il fd della socket, il buffer e la sua dimensione.

Notare come questa funzione sia sostanzialmente simile alla fullWrite(), solo che ovviamente avremo ora una nread invece di nwrite, e nleft rappresenta il numero di byte *da leggere* rimanenti.

Siccome stiamo leggendo, dobbiamo ancora prendere in considerazione la possibilità di incontrare un End Of File. Questa situazione si verifica quando nread = 0.
In tal caso, rompiamo il loop.

Prima della fine della funzione, impostiamo il puntatore a buffer a 0.