



LSM - a walk throughout penguin cops
Viaggio introduttivo al Linux Security
Module

Paolo Perego

`mailto:sponge@tiscali.it`

Introduzione

Affronteremo il problema del controllo degli accessi alle risorse di un sistema da parte dei processi utente, analizzando il framework LSM, in particolare:

- ⑥ la struttura della patch LSM;
- ⑥ l'integrazione del Linux Security Module con il kernel;
- ⑥ la struttura di un modulo *LSM compliant*;
- ⑥ vari esempi di utilizzo.

Stato dell'arte

- ⑥ Controllare gli accessi alle risorse di un sistema è un punto chiave per la messa in sicurezza del sistema stesso;
- ⑥ I sistemi per il controllo degli accessi si basano su politiche:
 - △ discrezionali;
 - △ mandatorie.
- ⑥ I meccanismi utilizzati dai moderni sistemi operativi si rivelano spesso inadeguati.

Quello che mi serve è...

- ⑥ Superare le limitazioni del sistema operativo in questo campo equivale a scrivere un sistema di controllo generico che garantisca:
 - △ la possibilità di implementare politiche di controllo differenti come parti a se stanti che si integrino col kernel;
 - △ la possibilità di integrare più politiche di controllo per ottenere comportamenti complessi;
 - △ una limitata invasività del sistema all'interno del kernel del sistema operativo.

Quello che mi serve è...

- ⑥ Linux offre il Linux Security Module, un framework che differenti progetti possono utilizzare per moduli che implementino politiche di controllo;
- ⑥ TrustedBSD nasce come versione di FreeBSD con l'aggiunta del framework MAC per la gestione delle politiche di controllo.

L'approccio prima di LSM

- ⑥ L'utente root è centrale nell'amministrazione della macchina:
 - △ una politica discrezionale garantisce qualsiasi privilegio;
 - △ ottenere i privilegi di root equivale ad avere il controllo dell'host.
- ⑥ Sistemi di protezione più evoluti dovevano essere pensati come moduli aggiuntivi al kernel.

L'approccio prima di LSM

- ⑥ Mancanza di un'interfaccia tra moduli ed *"internals"*;
- ⑥ Utilizzo dell'interposizione delle *"system call"* per accedere agli oggetti interni del kernel. Pericolo di:
 - △ scrivere del codice già presente nel kernel per replicarne le funzionalità;
 - △ creare eventuali patch *"ad hoc"* per poter accedere alle strutture dati interne;
- ⑥ Perdo i benefici del concetto di modularità.

Introduzione

- ⑥ LSM nasce nel 2001 con l'obiettivo di introdurre un framework comune a tutti i progetti riguardanti il controllo degli accessi;
- ⑥ Questo framework sarebbe dovuto essere:
 - △ leggero (le performance del sistema non devono degradare);
 - △ generico (è compito del modulo la definizione dei controlli sul sistema, LSM deve fungere solo da interfaccia);
 - △ in grado di garantire il supporto ad un modulo che implementasse le capability POSIX 1.e.

Introduzione

- ⑥ Si volevano superare i problemi legati all'interposizione delle chiamate di sistema e non si voleva rendere necessaria la scrittura di alcuna patch;
- ⑥ Vengono introdotti in punti strategici del codice degli hook che consentono di accedere alle strutture dati interne senza che il codice risulti eccessivamente invasivo rispetto al kernel.

La patch

- ⑥ LSM nasce prima come patch per i kernel 2.4.x mentre ora è incluso nella distribuzione del kernel stabile;
- ⑥ Vengono introdotti:
 - △ campi (*security fields*) nelle più importanti strutture interne del kernel;
 - △ hook in porzioni strategiche del kernel stesso;
 - △ meccanismi coi quali un modulo può registrarsi come *security module*.

I security fields

- ⑥ Definiti come puntatori (`*void`);
- ⑥ Usati per associare delle informazioni alla singola struttura dati interna (eg. posso voler sapere l'istante dell'ultimo controllo su una `task_struct`);
- ⑥ Gestiti completamente dal security module:
 - △ allocazione e deallocazione;
 - △ gestione dell'informazione associata al campo;
 - △ politica di gestione per gli oggetti del kernel esistenti prima del caricamento del modulo.

- ⑥ L'hook è un puntatore ad funzione implementata dal modulo *LSM compliant*;
 - △ esiste una funzione `dummy_*()` per ogni hook;
 - △ un modulo registra le proprie funzioni per gli hook di suo interesse;
 - △ il kernel chiama in modo trasparente le funzioni del modulo, se esistente, o le funzioni `dummy_*()`.
- ⑥ Gli hook disponibili sono contenuti nella `struct security_ops` in `include/linux/security.h`.

- ⑥ L'hook ha una natura restrittiva:
 - △ non posso fare un *override* delle decisione del kernel. Non darò l'occasione di controllare un accesso che di norma sarebbe vietato;
 - △ posso impedire degli accessi che sarebbero invece concessi dal kernel.
- ⑥ Esiste anche un minimo supporto per hook più permissivi.

- ⑥ Esempi di aree di interesse dove sono presenti degli hook:
 - △ ciclo di vita di un processo ed interazioni col kernel;
 - △ operazioni da eseguire prima e dopo l'esecuzione di un programma;
 - △ operazioni possibili sul singolo inode tra cui le operazioni di mount e unmount dei dispositivi;
 - △ ciclo di vita di una socket ed informazioni relative ai dispositivi di rete.

Kernel 2.4.19: `do_follow_link()`

```
static inline int do_follow_link(struct dentry *dentry, struct nameidata *nd)
{
    int err;
    if (current->link_count >= 5)
        goto loop;
    if (current->total_link_count >= 40)
        goto loop;
    if (current->need_resched) {
        current->state = TASK_RUNNING;
        schedule();
    }
    current->link_count++;
    current->total_link_count++;
    UPDATE_ATIME(dentry->d_inode);
    ...
}
```

Kernel 2.6.4: `do_follow_link()`

```
static inline int do_follow_link(struct dentry *dentry, struct nameidata *nd)
{
    int err = -ELOOP;
    if (current->link_count >= 5)
        goto loop;
    if (current->total_link_count >= 40)
        goto loop;
    cond_resched();
    err = security_inode_follow_link(dentry, nd);
    if (err)
        goto loop;
    current->link_count++;
    current->total_link_count++;
    update_atime(dentry->d_inode);
    ...
}
```

- ⑥ In assenza di un *security module* associati ad ogni hook ci sono delle funzioni che restituiscono sempre 0;
- ⑥ Una volta registrato il *security module* vengono copiati gli indirizzi delle funzioni handler che il modulo ha definito per uno o più hook;
- ⑥ In questo modo il kernel interroga in maniera trasparente il *security module* i cui handler implementano le politiche di sicurezza.
- ⑥ Alcuni hook sono posti in modo che il loro esito non influisca sul servizio offerto dal kernel (eg. hook per aggiornare i *security fields*).

Registrazione di un modulo

- ⑥ Durante il boot il kernel inizializza il framework attraverso la chiamata `security_scaffolding_startup()` (`security/security.c:56`):
 - △ controllo `dummy_security_ops` che contiene le funzioni standard associate agli hook di LSM;
 - △ associa le funzioni `dummy_*()` agli hook;
 - △ chiama eventuali funzioni di inizializzazione di codici che usano il framework ma che sono compilate staticamente nel kernel.

Registrazione di un modulo

- ⑥ Un modulo deve registrarsi con LSM utilizzando la funzione `register_security()` in `security/security.c:85`;
- ⑥ Questa funzione:
 - △ controlla la struttura `struct security_ops` passata come argomento;
 - △ controlla se esiste già un modulo registrato. Se esiste già un modulo attivo, la registrazione dovrà avvenire in altro modo;
 - △ imposta le funzioni dichiarate dal modulo come gestori degli *hook point* dichiarati dal framework.

Rimozione di un modulo

- ⑥ La rimozione di un modulo avviene attraverso la funzione `unregister_security()` in `security/security.c:115`;
- ⑥ Questa funzione:
 - △ controlla che si stia cercando di rimuovere un modulo effettivamente caricato;
 - △ ripristina le funzioni `dummy_*()` come gestori degli hook.
- ⑥ Durante la rimozione è compito del modulo deallocare i buffer negli oggetti interni del kernel.

Module stacking

- ⑥ Più *security module* possono essere caricati contemporaneamente;
- ⑥ Solo il primo modulo si registra direttamente con LSM attraverso `register_security()`;
- ⑥ Altri moduli possono essere caricati solo se il primo supporta lo *stacking* implementando:
 - △ sia `register_security()` che `unregister_security()`;
 - △ il codice necessario a chiamare gli handler dichiarati dai moduli impilati;
 - △ il codice che replica alcuni aspetti del framework stesso.

Module stacking

- ⑥ I moduli usano le funzioni `mod_reg_security()` e `mod_unreg_security` per interfacciarsi col modulo primario;
- ⑥ Dopo il controllo dei parametri vengono richiamate le funzioni `register_security()` e `unregister_security()` definite nel modulo primario;
- ⑥ È del modulo primario la scelta di permettere o non permettere lo stack di moduli;
- ⑥ È buona idea limitare la complessità che uno stack di security module introdurrebbe a livello di politiche di accesso.

Struttura di un modulo

Un modulo che vuole usare il nuovo framework deve:

- ⑥ implementare alcune funzioni che implementano le politiche di sicurezza;
- ⑥ dichiarare una `struct security_operations` facendo corrispondere ad ogni hook la funzione handler preposta;
- ⑥ durante il caricamento del modulo registrarsi presso il kernel o presso il security module principale;
- ⑥ durante la rimozione del modulo rimuovere la struttura dichiarata presso il kernel mediante le apposite funzioni.

Struttura di un modulo

- ⑥ Tutti gli hook disponibili sono dichiarati in `include/linux/security.h`;
- ⑥ Le funzioni handler dichiarate dal modulo devono rispecchiare i prototipi dichiarati da LSM;
- ⑥ Estendere le politiche di sicurezza equivale a scrivere nuovi handler per nuovi hook del framework.

Esempio di modulo: Nuso

- ⑥ Semplice modulo nato per dimostrare la flessibilità e la potenza di LSM;
- ⑥ Aumento la sicurezza del mio elaboratore impedendo che alcuni *uid* compiano determinate operazioni come:
 - △ eseguire determinate applicazioni (specificate a build time);
 - △ creare link sia *hard* che *soft*;
 - △ montare dispositivi;
 - △ creare file aventi privilegi di esecuzione oppure impostare gli stessi successivamente (`man chmod`).

Esempio di modulo: Nuso

- ⑥ Sviluppando il codice di `nuso` ci si accorge che le funzioni chiave sono:
 - △ `uid_lookup()`;
 - △ `prg_lookup()`.
- ⑥ Implementano semplicemente la ricerca dello *uid* del processo corrente tra quelli messi in “*quarantena*” e una funzione analoga per le applicazioni;
- ⑥ In un progetto reale il significato degli *uid* in quarantena andrebbe ribaltato: negare i privilegi a tutti tranne agli *uid* indicati.

Nuso: filtro sui programmi eseguiti

- ⑥ Intervengo prima che il kernel ricerchi un handler per il file binario scelto (`search_binary_handler()` in `fs/exec.c:1027`);
- ⑥ Se un uid tra quelli in quarantena cerca di eseguire un programma in quarantena occorre negare la richiesta;
- ⑥ Questo controllo è semplicistico:
 - △ un attacker potrebbe eseguire un link all'applicazione;
 - △ occorre controllare l'inode piuttosto che il nome del file;
 - △ per uso didattico questo primo controllo è sufficiente.

`nuso_bprm_check_security()`

```
static int nuso_bprm_check_security (struct linux_binprm *bprm) {  
#ifdef SUID_CHANGE_NOTIFY  
    if (current->uid != bprm->e_uid)  
        // notifica del cambio di privilegi  
#endif  
  
    if ((uid_lookup(current->uid) == 0) &&  
        (prg_lookup(bprm->filename) == 0)) {  
        // notifica dell'accesso negato  
        violations++;  
        return -1;  
    }  
    return 0;  
}
```

Nuso: filtro sulla creazione di link

- ⑥ Intervengo quando il layer del VFS sta cercando di creare il link (`vfs_link()` in `fs/namei.c:1832` nel caso di link hard e `vfs_symlink()` in `fs/namei.c:1761` nel caso di link simbolici);
- ⑥ Il controllo è solo sul codice dell'utente che sta chiedendo il servizio al kernel. Se è un utente in quarantena la nostra funzione handler negherà l'operazione;
- ⑥ Le tecniche di evadere dagli IDS che controllano il pathname del programma da proteggere falliscono.

Nuso: nuso_inode_link()

// La funzione nuso_inode_symlink() è assolutamente identica.

```
static int nuso_inode_link(struct dentry *old_dentry ,  
    struct inode *dir ,  
    struct dentry *new_dentry) {  
    if (uid_lookup(current->uid) == 0) {  
        // notifica dell'accesso negato  
        violations++;  
        return -1;  
    }  
    return 0;  
}
```

Nuso: filtro sul mount di un dispositivo

- ⑥ Intervengo prima che il dispositivo venga effettivamente montato dal kernel (`do_mount()` in `fs/namespace.c:775`;
- ⑥ Il controllo è sempre sul codice uid che ha richiesto l'operazione;
- ⑥ Le uniche operazioni che si compiono sui parametri passati al kernel servono per stampare informazioni di log coerenti col tipo di operazione che si vuole fare (remount, loopback mount, ...).

Nuso: filtro sulla creazione di file eseguibili

- ⑥ Intervengo appena prima che venga creato l'inode associato al file (`vfs_create()` in `fs/namei.c:1143`);
- ⑥ Controllo i flag coi quali si vuole creare il file (`man 2 open` per il significato del parametro `flags` usato per creare un file);
- ⑥ Nessuno tra proprietario, gruppo o tutti gli altri utenti deve possedere il permesso di esecuzione.

Nuso: nuso_inode_create()

```
static int nuso_inode_create (struct inode *dir ,  
    struct dentry *dentry , int mode) {  
  
    if (uid_lookup(current->uid) == 0) {  
        /*  
         * Quarantine user can't create file with exec permission.  
         */  
        if ((mode & S_IXUSR) || (mode & S_IXGRP) || (mode & S_IXOTH)) {  
            // notifica dell'accesso negato  
            violations++;  
            return -1;  
        }  
    }  
    return 0;  
}
```

Nuso: filtro sulla creazione di file eseguibili

- ⑥ Un file può essere creato eludendo i precedenti controlli;
- ⑥ Occorre impedire che possano essere impostati privilegi di esecuzione in un secondo momento;
- ⑥ Sfrutto l'hook `security_inode_setattr()` dichiarato in `notify_change()` in `fs/attr.c:130`;
- ⑥ Controllo gli attributi che si vogliono impostare ad un file e nego l'operazione se viola le politiche di sicurezza scelte.

Nuso: osservazioni

- ⑥ Scrivere un handler è semplice:
 - △ il prototipo della funzione è già fissato;
 - △ si hanno a disposizione gli oggetti interni al kernel;
 - △ si ha il pieno controllo sul comportamento del sistema.
- ⑥ Aggiungere una politica di sicurezza significa aggiungere un handler e controllare il comportamento del kernel in una certa occasione (es: creazione di socket, impostazione bit suid/sgid, ...);
- ⑥ L'impatto in termini prestazionali sul sistema dipende dalla complessità degli handler.

Conclusioni

- ⑥ LSM è potente ed è un'interfaccia semplice e ben strutturata;
- ⑥ È semplice scrivere moduli che implementano politiche di sicurezza;
- ⑥ LSM offre la possibilità di intervenire in maniera granulare sul comportamento del kernel;
- ⑥ Scrivere security module diventa più semplice e meno pericoloso per la stabilità del sistema.

Q&A

- ⑥ Domande?
- ⑥ Dubbi?
- ⑥ Commenti?

- ⑥ C. Wright, C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartman, “*Linux Security Modules: General Security Support for the Linux Kernel*”;
- ⑥ Linux Security Module project:
<http://lsm.immunix.org>;
- ⑥ NSA SELinux: <http://www.kernel.org>;
- ⑥ Nuso Security Module:
<http://www.sikurezza.org/angel>;
- ⑥ Sikurezza.org Project:
<http://www.sikurezza.org>.