SOA-project

Indice

Specifica: Monitor di Riferimento a Livello Kernel per la Protezione dei File

Questa specifica riguarda un Modulo del Kernel Linux (LKM) che implementa un monitor di riferimento per la protezione dei file. Il monitor di riferimento può trovarsi in uno dei seguenti quattro stati:

- OFF: significa che le sue operazioni sono attualmente disabilitate;
- ON: significa che le sue operazioni sono attualmente abilitate;
- REC-ON/REC-OFF: significa che può essere attualmente riconfigurato (sia in modalità ON che OFF).

La configurazione del monitor di riferimento si basa su un insieme di percorsi del file system. Ogni percorso corrisponde a un file/directory che al momento non può essere aperto in modalità scrittura. Quindi, ogni tentativo di aprire il percorso in modalità scrittura deve restituire un errore, indipendentemente dall'ID utente che tenta l'operazione di apertura.

Riconfigurare il monitor di riferimento significa che alcuni percorsi da proteggere possono essere aggiunti/rimossi. In ogni caso, cambiare lo stato attuale del monitor di riferimento richiede che il thread che sta eseguendo questa operazione sia contrassegnato con l'ID utente effettivo impostato su root, e inoltre la riconfigurazione richiede in input una password specifica del monitor di riferimento. Ciò significa che la versione crittografata della password è mantenuta a livello dell'architettura del monitor di riferimento per eseguire i controlli richiesti.

Spetta al progettista del software determinare se i suddetti stati ON/OFF/REC-ON/REC-OFF possono essere modificati tramite API VFS o tramite chiamate di sistema specifiche. Lo stesso vale per i servizi che implementano ciascun passaggio di riconfigurazione (aggiunta/rimozione di percorsi da controllare). Insieme alle operazioni a livello del kernel, il progetto dovrebbe anche consegnare codice/comandi dello spazio utente per invocare l'API di livello di sistema con parametri corretti.

Oltre alle specifiche sopra indicate, il progetto dovrebbe includere anche la realizzazione di un file system in cui un singolo file *append-only* dovrebbe registrare la seguente tupla di dati (per riga del file) ogni volta che viene tentato di aprire un percorso del file system protetto in modalità scrittura:

- l'ID del processo TGID
- l'ID del thread
- l'ID utente
- l'ID utente effettivo
- il percorso del programma che sta attualmente tentando l'apertura
- un hash crittografico del contenuto del file del programma

Il calcolo dell'hash crittografico e la scrittura della tupla sopra indicata dovrebbero essere eseguiti come lavoro differito.

Implementazione

Il progetto si compone di diversi moduli kernel che, cooperando, implementano le funzionalità richieste su versione 5.15 del kernel Linux:

- singlefile-FS
- Linux-sys_call_table-discoverer
- sys_call_installer
- reference-monitor-kprobes

Singlefile-FS

È stata implementata la funzione per la scrittura in append mode. Poiché è il kernel a dover scrivere sul file, è stata implementata la file operation `write_iter`, che viene internamente invocata dalla funzione `kernel_write()`. La funzione gestisce opportunamente anche le scritture che coinvolgono più blocchi.

È stata implementata anche la funzione per la file operation `write`, ma non è stata effettivamente *agganciata*: in questo modo, non sarà possibile scrivere sul file dallo spazio utente.

Facendo vari test durante lo sviluppo, nel file di log, ho trovato questa situazione:

```
TGID: 1786 PID: 1786 UID: 1000 EUID: 1000 prograTGID: 1786 PID: 1786 UID:
1000 EUID: 1000 program path: /usr/bin/zsh file content hash:
4ff9dbd18f4cad5234614cdf6a2b59e0a9a6326b09b8e86f40e1a55b5476d11d
m path: /usr/bin/zsh file content hash:
4ff9dbd18f4cad5234614cdf6a2b59e0a9a6326b09b8e86f40e1a55b5476d11d
```

Il risultato è stato riscontrato con le seguenti istruzioni dalla root del progetto:

```
$ user/add_path ../dir_prova
adding path: ../dir_prova
path added successfully
$ for ((i=0;i<100;i++)); do
```

```
echo "asd" > ../dir_prova/asd.txt
done
```

Poiché sembrava essere un problema di concorrenza, sono stati introdotti meccanismi di sincronizzazione basati su MUTEX nella funzione di scrittura. Il problema non si è mai più verificato nei test svolti.

Linux-sys_call_table-discoverer

Si tratta del modulo kernel fornito come esempio durante il corso e reperibile a questo link.

È stato tuttavia esteso per inizializzare una struttura di tipo `sys_call_helper_t`, struttura definita in `$PROJECT_ROOT/lib/sys_call_helper.h` nel seguente modo:

```
typedef struct _sys_call_helper_t {
    unsigned long **hacked_syscall_tbl;
    /* indexes to use starting from hacked syscall table to reach
ni_syscalls */
    int free_entries[MAX_FREE];
    /* how many free entries has been found? */
    int free_entries_count;
    /* which index was the last used to register a new syscall?? */
    int last_entry_used;
    /* register a new syscall: */
    int (*install_syscall)(unsigned long*);
    /* uninstall all syscalls: */
    void (*uninstall_syscalls)(void);
} sys_call_helper_t;
```

I puntatori alle due funzioni servono rispettivamente per registrare e cancellare nuove syscall dalla syscall table, e la loro implementazione è definita all'interno di questo modulo dalle funzioni:

```
int install_syscall(unsigned long *new_sys_call_addr);
void uninstall_syscalls(void);
```

L'istanza di questa struttura dati inizializzata viene poi esportata e verrà utilizzata nel modulo `sys_call_installer`, discusso successivamente.

Sys_call_installer

Questo modulo ha la responsabilità di installare e disinstallare syscall in maniera del tutto trasparente. Dopo aver recuperato gli indici delle entries utilizzabili dal modulo `Linux-sys_call_table-discoverer` che li memorizza nel campo `free_entries`, la struttura tiene traccia della posizione dell'ultimo indice utilizzato per installare le systemcall nel campo `last_entry_used`. Grazie a questo modulo, l'installazione e la rimozione di nuove systemcall diventa molto semplice, rendendo il software molto mantenibile: non servirà infatti specificare l'indice in cui memorizzarla, ma verrà identificato nell'indice dato da `free_entries[last_entry_used + 1]`.

Nel caso del progetto, le system call installate sono 5:

- `sys_add_path`: è la system call che si occupa di inserire un nuovo path tra quelli controllati dal reference monitor;
- `sys_rm_path`: è la system call che si occupa di rimuovere un path da quelli controllati dal reference monitor;
- `sys_change_monitor_state`: è la system call che si occupa di cambiare lo stato del reference monitor;
- `sys_get_paths`: è la system call che si occupa di inserire un nuovo path tra quelli controllati dal reference monitor;
- `sys_change_monitor_password`: è la system call che si occupa di cambiare la password del reference monitor;

Le ultime due non erano richieste nella traccia, ma sono state inserite perché credo che migliorino l'esperienza utente nell'interazione col software, senza considerare che sono state molto utili per il debugging durante lo sviluppo del software stesso.

All'interno del corpo delle system call definite in questo modulo si fa molto uso delle funzioni `copy_from_user()` e `copy_to_user()` per recuperare i parametri passati in user space. Poiché la maggior parte di questi parametri sono `char*`, si è utilizzata molto anche la funzione `strnlen_user`, con cui si può specificare anche la massima lunghezza consentita per una stringa utente.

Questo modulo definisce al suo interno un parametro di output: si memorizzeranno infatti su un file gli indici delle entries della syscall table utilizzati per installare le nuove funzioni. Questo è necessario per poter ottenere il numero delle syscall da invocare in user space in modo automatico, senza dover necessariamente cambiare a mano il valore di variabili o macro.

Questo modulo utilizza, come detto, una variabile esportata dal modulo `Linux-sys_call_table-discoverer`, quindi dovrà essere montato successivamente e smontato precedentemente rispetto al modulo da cui dipende.

reference-monitor-kprobes

Questo modulo è il *core* del progetto. Per poter filtrare operazioni di scrittura su file specificati, è stato utilizzato il meccanismo delle `kprobes` per aggiungere definire dei wrapper a diverse funzioni che realizzano scritture. In particolare, le funzioni che sono state filtrate sono:

- `do_filp_open`: per filtrare le operazioni di scrittura su file
- `do_unlinkat`: per filtrare comandi per la rimozione di file, come `rm`
- `do_rmdir`: per filtrare comandi per la rimozione di directory, come `rmdir`
- `do_mkdirat`: per filtrare comandi per la creazione di directory, come `mkdir`
- `do_renameat2`: per filtrare comandi per la modifica del path, come `mv`

La scelta di filtrare queste funzione è dettata dal fatto che tutte prendono in input una `struct *filename`, da cui è possibile ricavare il path del file. Da questo path è possibile ricavare la relativa `struct path` sfruttando la funzione `kernel_path()` con cui si *filla* una `struct path` al cui interno è possibile trovare un riferimento alla `dentry`.

In questo modo, quindi, è possibile avere una uniformità di soluzione per il recupero della `dentry` del file, che è una struttura *core* all'interno del progetto.

Il modulo utilizza le struttura dati definite in `$ROOT_PROJECT/reference-monitor-kprobes/lib/reference_monitor.h`:

```
typedef struct _reference_monitor_t {
    unsigned char state;
    // this will keep hashed pass
    char hashed_pass[HASH_SIZE];
    spinlock_t lock;
    struct dentry **filtered_paths;
    int filtered_paths_len;
    int (*add_path)(const char *new_path);
    int (*rm_path)(const char *path);
    char* (*get_path)(int index);
    int (*set_state)(unsigned char state);
} reference_monitor_t;

typedef struct _deferred_work_t {
    pid_t tgid;
    pid_t pid;
    uid_t uid;
    uid_t euid;
    char command_path[MAX_PATH_LEN];
    // since it will be stored on a file, i will trace the hex
representation of the hash
    char command_hash[HASH_SIZE * 2 + 1];
    struct work_struct the_work;
} deferred_work_t;
```

Si tratta delle strutture dati che vengono utilizzate rispettivamente per la gestione del reference monitor e del *defferred work*.

reference_monitor_t

Come si vede dalla definizione della struttura, il reference monitor mantiene una lista di `dentry *` che puntano agli `inode` dei file filtrati. La scelta di utilizzare le `dentry` piuttosto che gli `inode` stessi è dettata dalle seguenti motivazioni:

- a partire da una `dentry`, è molto facile risalire l'albero del filesystem per:
    - recuperare l'`inode` della directory parent e verificare se la cartella parent risulta filtrata. Infatti, in questo caso, tutte le scritture all'interno di nodi figli di quella directory dovranno essere filtrate.
    - recuperare il nome della directory parent, necessaria per ricostruire il path completo del file di cui necessita la systemcall `get_paths`
- passare dalla `dentry` al relativo `inode` è un'operazione molto semplice, mentre il viceversa può risultare molto complicato

Nonostante la struttura dati tenga traccia delle `dentry`, il controllo fatto per verificare se un file è filtrata avviene confrontando gli `inode` puntati dalle dentry stesse con l'inode del file che si sta cercando di

modificare.

Il campo `state` della struttura serve invece a tenere traccia dello stato in cui si trova attualmente il reference monitor. È definito come un `unsigned char` poiché, in effetti, basterebbero soltanto due bits per differenziare i quattro stati possibili.

Nel file `$ROOT_PROJECT/lib/reference_monitor_states.h` sono definite le seguenti costanti:

```
#define OFF     0x0 // 00
#define ON      0x1 // 01
#define RECOFF  0x2 // 10
#define RECON   0x3 // 11

// last bit used to check if state is valid
#define INVALID_STATE  1 << 7
```

Oltre alla definizione dei quattro stati, è stato aggiunto l'`INVALID_STATE`: l'ultimo bit impostato a 1 identificherà uno stato non valido e che qualcosa è andato storto.

È anche presente uno `spinlock`, con cui si sincronizzano le operazioni sulla stessa struttura dati.

L'istanza di questa struttura utilizzata nel modulo viene anche esportata: verrà infatti usata dal modulo `sys_call_installer` nel corpo delle system call lì definite.

deferred_work_t

È la struttura dati che viene inizializzata quando si filtra una scrittura. Contiene il campo `struct work_struct the_work` che verrà inserito in una `work_queue` tramite la macro: `__INIT_WORK()`. Alla funzione specificata viene passata proprio questa istanza di `work_struct`, e per risalire alla struttura `deferred_work_t` in cui è contenuta è stata utilizzata la macro `container_of()`.

user space

Come da traccia, sono definiti anche i sorgenti e il Makefile per la generazione degli eseguibili utenti. Questi eseguibili useranno però delle system call nuove, la cui indicizzazione non è fissa. Si sfrutta quindi lo script `$ROOT_PROJECT/generate_syscall_numbers_header.sh`. Questo script legge il parametro di output del modulo `sys_call_installer`, lo analizza per filtrare quelli diversi da 0, e definisce una stringa composta da una serie di `#define` per definire il numero delle system call che verranno poi usate user space. Questa stringa verrà poi salvata all'interno del file `$ROOT_PROJECT/user/lib/sys_call_numbers_header.h`. Un esempio di file di output è il seguente:

```
#define SYS_ADD 134
#define SYS_GET 156
#define SYS_RM 174
#define SYS_CHANGE_PASS 177
#define SYS_CHANGE_STATE 178
```

Questo header file verrà poi importato dove necessario per effettuare le invocazioni alle syscall corrette.

Nella cartella `ROOT_PROJECT/user` sono definiti tutti i file sorgenti per gli eseguibili che invocheranno le systemcall. In realtà, questi sorgenti si occupano solo della gestione dei parametri passati da riga comando. Includono però l'header file `system_calls.h`, definito in `ROOT_PROJECT/user/lib` insieme al relativo file sorgente da cui generare il `.o` che deve essere linkato in fase di compilazione. In questo header file sono definite le funzioni che invocano di fatto la systemcall opportuna.

Affinché i file eseguibili siano utilizzabili per interagire col reference monitor, devono essere lanciati con `euid` pari a 0. Riguardo questo aspetto, la traccia specificava:

> Reconfiguring the reference monitor means that some path to be protected can be added/ removed. In any case, *changing the current state of the reference monitor* requires that the thread that is running this operation needs to be marked with effective-user-id set to root, and additionally the reconfiguration requires in input a password that is reference-monitor specific.

ma non era molto chiaro se per *current state* si intendeva lo stato della struttura (ON/OFF/RECON/RECOFF) o era inteso anche a livello di riconfigurazione: aggiungere un path, ad esempio, porta il reference monitor in un nuovo stato. Si è deciso di seguire quest'ultima strada: ogni eseguibile, per non fallire, deve essere lanciato dall'utente root.

Manuale di installazione

Per semplificare e automatizzare la compilazione dei moduli, il loro montaggio e la generazione dei file eseguibili utente è stato pensato lo script `load.sh`. Questo script compila i vari moduli tenendo conto di eventuali dipendenze da esse, genera l'header file usato lato utente per gli indici delle system call e poi compila i binari all'interno della cartella `$ROOT_PROJECT/user/bin`.

Dualmente, lo script `unload.sh` si occupa dello smontaggio e dell'eliminazione dei file di output dei vari processi.

Entrambi gli script sfruttano i Makefile e gli script di `(un)load.sh` interni ai vari moduli.

Gestione degli imports

Nella cartella `$ROOT_PROJECT/lib` sono presenti gli header files che sono condivisi tra più moduli e/o lo spazio utente. In `$ROOT_PROJECT/$MODULE_ROOT/lib`, invece, ci sono gli header files che sono relativi al singolo modulo; stesso discorso vale per `$ROOT_PROJECT/user/lib`.