SVILUPPO DI AGENTI PER IL RILEVAMENTO DI INTRUSIONI TRAMITE INTROSPEZIONE

Fabio Campisi

INDICE

FIGURE

---

1. Introduzione

Negli anni, i *malware*, e i *rootkit* [9,40] in particolare sono diventati sempre più sofisticati, per cercare di eludere i sempre più sviluppati ed efficaci sistemi di sicurezza. I primi rootkit erano quelli *user-level*, che modificavano i comandi di sistema come *ls* o *ps* per nascondere all'amministratore del sistema la presenza di alcuni processi dell'attaccante. Questi rootkit sono facili da individuare e non provocano danni rilevanti ai sistemi, poiché non modificano alcuna parte del kernel. Ben presto gli attaccanti hanno utilizzato i più pericolosi rootkit *kernel-level* [11,24], in grado di compromettere il kernel del sistema. Essi operano perciò al livello più basso possibile ed hanno delle potenzialità illimitate. Per nascondere la loro presenza sul sistema, agiscono come i rootkit user-level, ma modificano lo stato del kernel, rendendo più complessa la loro individuazione e possono rilevare la presenza di strumenti anti-rootkit sul sistema e disabilitarli. In pratica, essi consentono ad un attaccante di impadronirsi del livello più basso del sistema e di fornire all'amministratore dati manipolati sullo stato dello stesso, trasmettendo un falso senso di sicurezza.

Chiaramente, conoscendo le tecniche degli attaccanti, gli strumenti sviluppati per difendere il sistema [10,31] hanno migliorato le loro capacità di rilevare i kernel-rootkit, ma anche quest'ultimi si sono a loro volta migliorati [38]. Assistiamo così ad una "lotta" infinita tra amministratori ed attaccanti, per prevenire le mosse del rispettivo avversario ed ottenere il pieno controllo del sistema.

La radice del problema è che l'architettura di molti sistemi operativi, tra i quali Linux, è tipicamente *monolitica*; questo significa che tutto il codice del kernel si trova nello stesso *dominio di protezione*, e qualsiasi funzione nel kernel, inclusi i kernel-rootkit e i tool anti-rootkit, può leggere e modificare una qualsiasi posizione in tutto lo spazio kernel. Esistono dei metodi che, tenendo conto di questa situazione, cercano di individuare i kernel-rootkit tramite l'analisi del flusso di esecuzione delle componenti del kernel [8,20]. In aggiunta alle tecniche di rilevamento delle intrusioni/attacchi, alcuni metodi [24] propongono delle tecniche per il ripristino automatico dello stato corretto del sistema. I *Network-IDS* [43] analizzano il traffico in transito su un sottoinsieme di una rete in cui risiede un insieme di host, alla ricerca di anomalie quali, ad esempio, firme di attacchi noti. In questo modo, essi monitorano gli host dall'esterno e non possono essere compromessi da attacchi portati agli host. Per la stessa ragione, hanno delle limitazioni in

termini di visibilità e azione rispetto alle tecniche di controllo locale, come quelle implementate dagli *Host-IDS* [43] e per questo risulta relativamente facile eluderli.

I possibili approcci per risolvere il problema sono due e cioè per separare il dominio su cui sono eseguiti i controlli da quello dove è eseguito il sistema, senza però perdere visibilità dello stato interno del sistema.

- Soluzione *hardware-based*: il sistema viene esteso mediante unità hardware speciali, per esempio una scheda PCI, che è completamente indipendente dal sistema operativo e possiede una propria CPU. Questa CPU utilizza un accesso DMA (*Direct Memory Access*) per analizzare periodicamente la memoria del sistema alla ricerca di rootkit. [1,32,47]

  Questa è una soluzione in grado di garantire un alto livello di sicurezza al sistema monitorato, ma impone dei notevoli vincoli hardware sulla sua realizzazione. Negli ultimi anni, sono state sviluppate nuove tecnologie per renderla utilizzabile su larga scala [17].

- *Virtualizzazione*: è una soluzione software al problema che prevede la creazione di un ambiente di esecuzione, all'interno del quale viene emulato il comportamento delle risorse presenti ad un certo livello del sistema, in particolare quelle hardware/firmware. L'ambiente creato è detto *macchina virtuale* (VM). Questa tecnologia introduce un nuovo livello nel sistema, che permette di monitorare in ogni momento lo stato interno di una VM da un dominio di protezione diverso ed indipendente. [13,19,35,49]

  Questa soluzione non offre lo stesso livello di separazione di quella hardware-based, ma può essere implementata in modo più flessibile ed, inoltre, offre un insieme di benefici per altri aspetti oltre a quello della sicurezza.

Il lavoro svolto e descritto in questa tesi è stato sviluppato nell'ambito del progetto *Psyco-Virt* [37], che sfrutta il secondo tipo di soluzione. Psyco-Virt fonde le tecniche standard per il rilevamento di intrusioni su sistemi distribuiti e le tecniche di virtualizzazione. Esso prevede di eseguire l'IDS (*Intrusion Detection System*) al livello del software di virtualizzazione che, in questo caso, emula il funzionamento delle risorse hardware (processore, memoria, dispositivi di I/O) e crea e gestisce le macchine virtuali, ognuna delle quali esegue il sistema operativo. Questo tipo di virtualizzazione offre la possibilità di implementare l'*introspezione* [13,37] dell'OS, eseguito da una VM.

L'introspezione è una tecnica che prevede di analizzare la memoria di una macchina, nel nostro caso virtuale, durante l'esecuzione di applicazioni, alla ricerca di tracce relative ad intrusioni e attacchi. Questa tecnica sarà descritta nel prossimo capitolo.

Nel seguito della tesi saranno descritte le caratteristiche di *Xen* [37,44], il software di virtualizzazione utilizzato e quelle del sistema Psyco-Virt. Verrà quindi descritto il lavoro svolto, che consiste nell'implementazione della libreria di introspezione chiamata *Xen-VMI* e nella definizione di un insieme di nuovi controlli, integrati con quelli già presenti nel sistema Psyco-Virt. Nella sezione finale sarà discussa l'efficacia e l'efficienza delle nuove componenti del sistema.

---

2. Introspezione

Questo capitolo descrive le principali caratteristiche dell'introspezione, l'aspetto centrale del lavoro svolto e che è alla base di tutti i controlli sviluppati. L'introspezione è una tecnica di rilevazione delle intrusioni basata sulla valutazione di un insieme di condizioni logiche, definite in termini della memoria di un sistema. Queste condizioni possono essere dedotte, in maniera automatica, da uno strumento che analizza il software in esecuzione oppure stabilite dall'amministratore in base a proprie conoscenze.

Per quanto riguarda il lavoro descritto nel seguito, l'introspezione è implementata mediante la *virtualizzazione* [37], una tecnologia che permette di incapsulare lo stato completo di un sistema di elaborazione all'interno di una macchina virtuale. Se assumiamo che il sistema sia visto come una composizione di macchine virtuali, strutturata su più livelli, a partire dal livello hardware/firmware, fino ad arrivare a quello delle applicazioni, possiamo considerare la virtualizzazione come un nuovo livello di astrazione dei sistemi. Ogni livello, o macchina virtuale, fornisce un'astrazione delle risorse ai livelli inferiori, rendendone più semplice l'utilizzo.

Fig.1: Struttura a livelli dei computer

La virtualizzazione si inserisce tra due dei livelli di Fig.1; nel nostro caso, vengono virtualizzate le risorse del livello hardware/firmware, cioè quelle gestite dai sistemi operativi. La proprietà più importante, dal punto di vista della sicurezza, è che il livello introdotto permette di monitorare in ogni momento lo stato di una macchina virtuale in esecuzione, cioè di implementare l'introspezione a "run-time".

Chiaramente, esiste un “gap semantico” [19] tra la visione del sistema che si può avere dall’interno e quella che l’introspezione permette dall’esterno. Dall'interno del sistema, si ha una visione ad alto livello, basata su entità come processi, file, moduli del kernel. Dall’esterno, invece, gli elementi base sono pagine di memoria, blocchi del disco e registri del processore. Per questo motivo, è necessario ricostruire le strutture dati che rappresentano le varie parti del sistema di cui interessa verificare la consistenza. L'obiettivo è quello di mantenere la stessa visione del sistema permessa da un controllo locale, in stile Host-IDS. Il controllo sull'host però, viene effettuato dall’esterno del sistema, al livello del software di virtualizzazione, in un dominio di protezione diverso da quello in cui viene eseguito il sistema dell'host.

Nel seguito supponiamo che il software di virtualizzazione sia sicuro, e quindi non possa essere compromesso. I motivi di questa assunzione saranno discussi nella prossima sezione. Sotto questa ipotesi, l’amministratore ottiene un importante vantaggio sugli attaccanti, perché essi non possono eludere la sorveglianza dei controlli implementati con l'introspezione. Infatti, i controlli sui dati sono eseguiti da un dominio di protezione sicuro e diverso da quello in cui viene eseguito l'OS, su cui l'attaccante agisce con i rootkit kernel-level. L’assunzione iniziale è quindi molto importante, poiché permette di includere nella *Trusted Computing Base* (TCB) tutte le parti critiche del sistema, controllate tramite introspezione (vedi sezione 2.2).

Questi sono i vantaggi, in termini di sicurezza, che offre questa tecnologia:

- Per un attaccante è più difficile modificare i dati che vengono analizzati tramite introspezione, poiché può compromettere lo stato del sistema in esecuzione sulla macchina virtuale, ma non lo stato visibile al livello inferiore della virtualizzazione.
- Confrontando i dati ottenuti tramite introspezione con quelli ottenuti dall’host monitorato, tramite le normali tecniche di rilevamento delle intrusioni, è possibile scoprire la presenza di un attaccante che ha modificato alcune parti del sistema per nascondere le sue tracce.
- Si possono controllare gli agenti del sistema, cioè le componenti del sistema di rilevamento delle intrusioni eseguite sulle VM. Ad esempio, è possibile monitorare le pagine di memoria, che contengono le istruzioni dei processi associati agli agenti, verificando che non siano alterate.

Nel seguito di questo capitolo sarà descritto in maniera approfondita Xen, il software di virtualizzazione utilizzato dal sistema Psyco-Virt, per poi esaminare le caratteristiche dell'architettura e il funzionamento dello stesso sistema Psyco-Virt, nell'ambito del quale è stato svolto il lavoro oggetto di questa tesi.

2.1 Xen

Xen [37,44] è il software che implementa una virtualizzazione delle risorse a livello hardware, interponendosi tra il livello hardware/firmware e quello del sistema operativo. Il nuovo livello ha il compito di astrarre ogni componente di una macchina fisica, quali il processore, la memoria e i dispositivi di I/O, creando degli ambienti di esecuzione software che emulano il comportamento dell'hardware della macchina. Questo compito è affidato al *Virtual Machine Monitor (VMM)* [44], detto anche *Hypervisor*, che si occupa di creare e gestire l'esecuzione concorrente delle VM sulla stessa macchina reale, garantendo: (1) la compatibilità di tutte le applicazioni che possono essere eseguite sulla macchina reale, (2) l'isolamento tra le VM, per motivi di correttezza e sicurezza e (3) un buon grado di efficienza rispetto a quella del sistema reale.

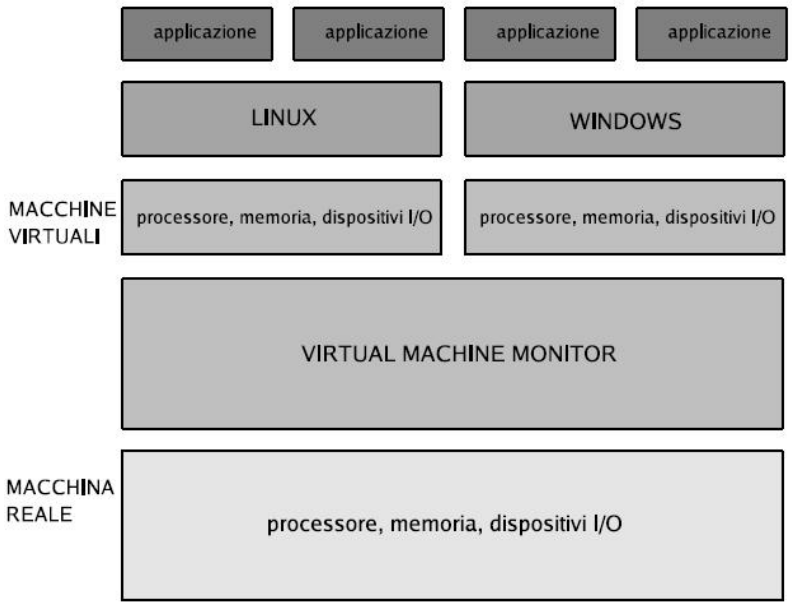


Fig.2: Struttura a livelli con virtualizzazione

Il VMM è da considerarsi *trusted* perché: (1) implementa un'interfaccia molto più semplice di quella di un sistema operativo che, ad esempio, implementa qualche centinaio di chiamate di sistema e quindi ha maggiori vulnerabilità; (2) è implementato con un

numero ridotto di linee di codice rispetto ad un OS, quindi è più facile verificare formalmente la correttezza della sua implementazione. Di conseguenza è meno probabile che esso presenti delle vulnerabilità.

Il software eseguito sulle macchine virtuali, che nella terminologia di Xen sono chiamate *domini*, si può comportare come se fosse eseguito direttamente sull'architettura fisica e non deve essere modificato per essere eseguito sulle VM. In questo modo, il VMM risulta completamente trasparente al livello software dei domini. In realtà, è solo il VMM che ha accesso diretto alle risorse hardware; le macchine virtuali vi accedono utilizzando l'interfaccia messa loro a disposizione. Inoltre, la proprietà dell'incapsulamento garantisce che tutto il software sia eseguito dalla singola macchina virtuale, cioè garantisce l'isolamento tra i domini permettendone anche un'esecuzione concorrente.

Infine, Xen fornisce un insieme di funzionalità per implementare politiche *MAC* (*Mandatory Access Control*), per il controllo degli accessi alle risorse condivise tra le VM e in generale, per mediare tutte le operazioni critiche per la sicurezza nell'ambito della virtualizzazione.

La performance effettiva di un sistema incapsulato in una macchina virtuale è molto vicina a quella reale; questo perché l'interfaccia verso le risorse hardware, fornita dal VMM, è uguale o molto simile a quella reale fornita dal sistema operativo. Di conseguenza è facile mappare le richieste da un'interfaccia all'altra. Inoltre, il VMM viene invocato solamente quando devono essere eseguite le istruzioni privilegiate, che permettono di controllare e gestire i domini, ad esempio di effettuare lo scheduling dei domini. In tutti gli altri casi, la macchina virtuale, quando è in esecuzione, accede alle risorse in maniera diretta, ed esegue le istruzioni direttamente sulla macchina reale.

Se, come detto in precedenza, la virtualizzazione è trasparente per il software applicativo, vi possono essere invece alcune differenze tra l'interfaccia verso le risorse hardware, offerta dal sistema operativo e quella implementata dal VMM. Le differenze dipendono dal tipo di virtualizzazione implementata e poiché nel caso di Xen la virtualizzazione è parziale, (*para-virtualization*), sono necessarie alcune modifiche al codice del sistema operativo, in modo che possa essere eseguito su una macchina virtuale. Questa è un'importante differenza tra Xen e i software che usano la "full virtualization", ad esempio, VMware [46].

Descrizione dettagliata

Durante l'inizializzazione di Xen, viene creato un dominio iniziale, responsabile della gestione e del controllo di Xen, detto *dominio 0*. Questo dominio è "privilegiato" rispetto agli altri, detti domini *guest*, perché dispone di un'interfaccia di controllo, che permette di:

- creare e distruggere i domini guest;
- specificare diversi parametri di esecuzione per i domini;
- creare interfacce di rete virtuali.

L'interfaccia "para-virtualizzata" può essere suddivisa in tre componenti principali: la gestione della memoria, la CPU e i dispositivi di I/O. È importante che la gestione garantisca proprietà di compatibilità ed isolamento, mantenendo un alto livello di efficienza.

La gestione della memoria è cruciale per l'efficienza; Xen realizza due diversi tipi di spazio di indirizzamento: uno contenente gli indirizzi della *machine-memory*, l'altro quelli della *pseudo-physical memory*. La "machine memory" è la memoria fisica presente sulla macchina e include tutta la memoria associata ai domini, quella usata da Xen e quella non allocata. La "pseudo-physical memory" è un'astrazione che permette di offrire ai sistemi operativi uno spazio di memoria logico contiguo, anche se esso in realtà è sparso. È stata adottata questa astrazione perché molti sistemi operativi richiedono che le pagine fisiche siano contigue. Per implementare questa astrazione, si utilizza una tabella globale, che mappa gli indirizzi appartenenti alla "machine memory" in quelli della "pseudo-physical memory", e una tabella in ogni dominio per implementare la mappatura inversa.

In molte architetture, ad esempio la 386, la protezione è basata sul concetto dei livelli di privilegio, detti anche *rings*. Le istruzioni che un modulo può eseguire dipendono dal rings in cui viene eseguito. In questo modo, i rings permettono di limitare l'esecuzione di alcune operazioni da parte, ad esempio, delle applicazioni. Nell'architettura 386 i rings sono 4, lo 0 è quello con più privilegi, il 3 quello con minor privilegi; solitamente l'OS viene eseguito nel ring 0 e le applicazioni sul 3. Xen richiede che gli OS vengono modificati per essere eseguiti sul ring 1; in questo modo tutte le operazioni privilegiate sono eseguite solo da Xen, eseguito al livello 0. Se un sistema operativo tenta di eseguire un'istruzione privilegiata, il processore genera un'eccezione che trasferisce il controllo a Xen. Per questo motivo, la tabella per la gestione delle interruzioni ed eccezioni di ogni

OS, è sostituita da una *virtual IDT*, che contiene i puntatori ai gestori definiti ed eseguiti all'interno del dominio 0, da Xen.

Per quanto riguarda l'implementazione e l'uso dei dispositivi di I/O, Xen offre due tipologie di driver:

- il *fontend driver*, che viene eseguito sui domini guest
- il *backend driver*, che viene eseguito sul dominio 0, oppure da un driver specifico che ha accesso diretto all'hardware.

Il "frontend driver" fornisce all'OS guest un'interfaccia per accedere al dispositivo uguale a quella reale, cioè a quella definita dal driver originale. Il driver riceve le richieste dal sistema operativo e invia una richiesta al "backend driver", che deve soddisfare la richiesta di I/O. Il "backend driver" deve controllare se le richieste sono corrette, se non sono in contrasto con i principi di isolamento e se rispettano la politica di sicurezza, e solo in questo caso trasmetterà le richieste al dispositivo hardware. Quando il dispositivo ha terminato le operazioni, il "backend driver" segnala al "frontend driver" che i dati sono pronti e, successivamente, il "frontend driver" lo segnalerà al sistema operativo. Lo scambio di messaggi tra i due driver, è implementato mediante la tecnica della memoria condivisa, utilizzata, in generale, per implementare lo scambio di messaggi tra un dominio e un altro.

2.2 Psyco-Virt

Il progetto Psyco-Virt [37] integra la tecnologia della virtualizzazione con le normali metodologie di rilevamento delle intrusioni su sistemi distribuiti, per ottenere un'architettura in grado di rilevare e prevenire le intrusioni monitorando i sistemi a più livelli. Si sfrutta così la passibilità offerta dal VMM di ispezionare in maniera diretta lo stato dell'host che si vuole monitorare, installando l'*IDS* [37,43] al suo interno. In questo modo, l'IDS ha la stessa visibilità dell'host che avrebbe se fosse installato sullo stesso, come avviene per gli Host-IDS. Esso è però più resistente agli attacchi, poiché non fa parte dell'host, ma si trova al livello inferiore, nel VMM (vedi Fig.3). Ovviamente, le assunzioni di maggior sicurezza sono giustificate se si assume che un VMM sia più difficile da attaccare e compromettere rispetto ad un host normale (vedi sezione 2.1). Nel seguito assumiamo che il VMM sia *trusted*.

L'unione dei due sistemi, VMM e IDS, produce la tecnologia della *Virtual Machine Introspection* (VMI) [13,37], che sta alla base dello sviluppo di tutti i controlli realizzati sull'architettura in questione, ed oggetto del lavoro svolto.

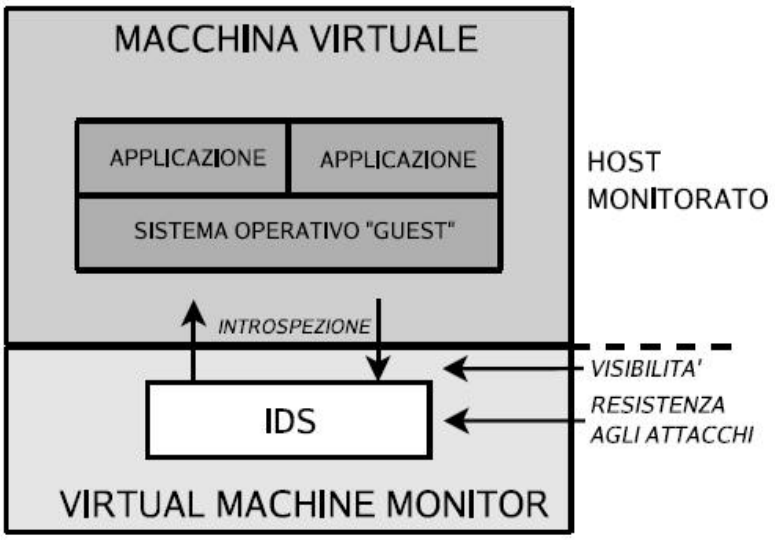


Fig.3: Introspezione di una VM

L'architettura del sistema è distribuita e prevede un insieme di macchine virtuali allocate su un insieme di nodi reali. Ogni nodo reale esegue un VMM, che applica l'introspezione sulle VM "locali". Per semplicità di implementazione l'IDS viene eseguito dal *dominio 0* di Xen che può essere considerato come un'estensione del VMM. Ogni VM monitorata esegue un insieme di *agent* (o agenti), coordinati da un *manager*, che implementa le funzionalità di un Network-IDS ed ha accesso all'interfaccia di controllo esportata dal Virtual Machine Monitor. Il manager deve eseguire due compiti:

- esaminare lo stato delle macchine virtuali tramite introspezione;
- ricevere ed analizzare tutti i messaggi relativi a tentativi di intrusione, inviati dalle componenti del sistema installate sulle macchine virtuali (agenti).

Quando il manager rileva delle intrusioni su un host monitorato, esegue delle azioni in risposta a tali eventi, agendo sullo stato di esecuzione del dominio attaccato. Elenchiamo alcune delle azioni possibili:

- bloccare l'esecuzione del domino attaccato e salvare il suo stato su un file;
- terminare i processi eseguiti dall'attaccante;
- chiudere la connessione della VM con la rete locale;

- disconnettere l'attaccante connesso con la VM;
- inviare un segnale alla console di amministrazione.

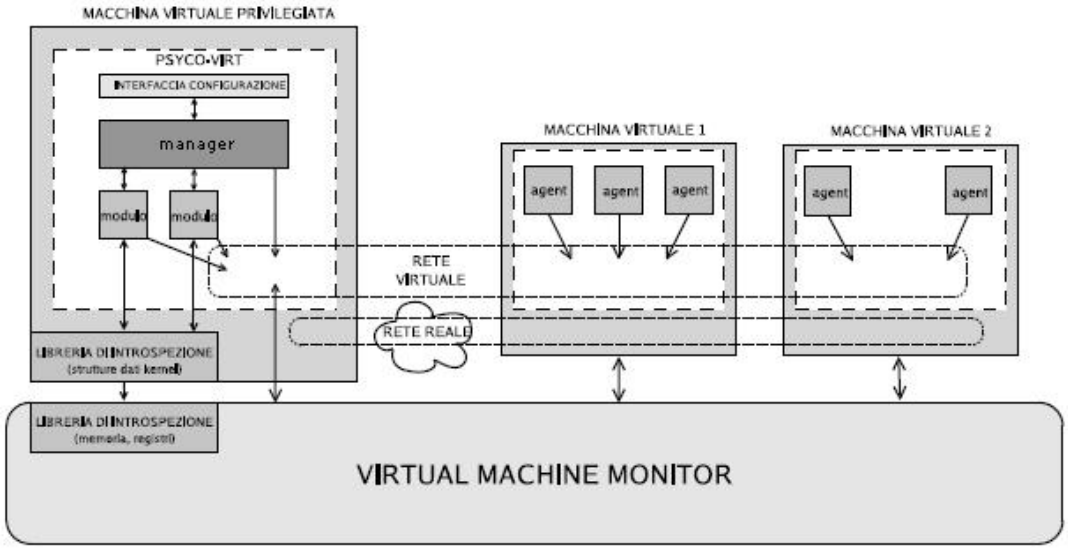


Fig.4: Architettura di Psyco-Virt

Dal dominio 0, il manager può esaminare lo stato di ogni macchina virtuale, ed in particolare le parti critiche per la sicurezza dei sistemi. Questo compito è facilitato dalla libreria di introspezione di "alto livello", che permette di interpretare ed utilizzare i dati di "basso livello", cioè quelli forniti attraverso la libreria di controllo di Xen. Questa libreria permette di accedere al contenuto della memoria, dei blocchi del disco e dei registri del processore. I controlli in introspezione si occupano di verificare la consistenza delle parti critiche del nucleo (kernel) del sistema e garantiscono la corretta esecuzione degli agent, i quali, a loro volta, effettuano un insieme di controlli sui processi a livello utente. La corretta esecuzione degli agent viene garantita mediante controlli di due tipi:

- Quelli implementati da moduli, eseguiti dal manager, che recuperano i dati critici mediante introspezione e, utilizzando la libreria di introspezione di "alto livello", li interpretano, per poi confrontarli con i dati forniti dagli agent.
- Quelli che verificano che le pagine relative alla *text area* dei processi associati agli agent non siano modificate.

In questo scenario, gli agent si comportano come dei normali Host-IDS: rilevano intrusioni e attacchi sul sistema e informano il loro coordinatore (il manager), che deciderà cosa fare.

Le comunicazioni tra manager e agent avvengono su una rete "virtuale", creata dal dominio 0, tramite OpenVPN [30], e detta *rete di controllo*. Il manager, oltre a ricevere le comunicazioni dai propri agent, può trasmettere loro delle richieste, per ottenere altre informazioni, nel caso si presentino situazioni che non riesce a valutare. Inoltre, i manager possono comunicare tra loro scambiandosi informazioni di controllo, per combattere meglio gli attacchi distribuiti. La rete di controllo può essere utilizzata solo dalle componenti del sistema Psyco-Virt (manager e agenti), e non è raggiungibile da nessun host. Un opportuno controllo *anti-spoofing* è stato introdotto per verificare che gli indirizzi sorgenti dei pacchetti, inviati sulla rete, siano tra quelli autorizzati.

Infine, Psyco-Virt è dotato di un'interfaccia di configurazione, tramite la quale l'amministratore di sistema può avviare e sospendere Psyco-Virt, specificando i parametri di esecuzione del sistema, che determinano:

- quali controlli effettuare;
- quali macchine virtuali monitorare;
- i livelli di logging, che identificano la gravità associata ad un evento;
- le azioni da eseguire, in risposta ad ogni evento che può essere rilevato.

Nelle VM "guest" è installato il sistema operativo Linux, distribuzione Debian [5], mentre sul dominio 0 la distribuzione Fedora [6] e nel seguito, quando si parlerà di aspetti relativi agli OS, il riferimento a Linux sarà sottointeso.

In sintesi, si può affermare che la sicurezza del sistema è garantita da un approccio a due livelli:

1) verifica, tramite introspezione, dell'integrità delle parti critiche del sistema eseguito sulle VM monitorate;
2) estensione del singolo sistema operativo con un insieme di funzioni di sicurezza, tra cui gli agenti, che eseguono una serie di controlli sull'integrità di altre parti del sistema e delle componenti a livello utente, come processi e file.

Esistono alcuni sistemi operativi che implementano in modo nativo il secondo step, ad esempio SELinux [39]. Il vantaggio di Psyco-Virt è che utilizza le normali tecniche di rilevamento delle intrusioni sulle macchine virtuali, sulle quali può essere applicata la tecnica della VMI, che permette di effettuare un insieme di controlli non eludibili dagli attaccanti, poiché si assume che l'introspezione faccia parte del *Trusted Computing Base*

(TCB). Sotto queste ipotesi l'approccio considerato riesce a garantire l'integrità di tutte le componenti critiche del sistema, effettuando i controlli a più livelli. L'introspezione verifica l'integrità delle parti del sistema che sono cruciali per garantire la corretta esecuzione delle funzioni di sicurezza (tra cui gli agenti). Grazie a questi controlli, il TCB viene esteso e comprende anche le componenti eseguite sulle VM monitorate e, di conseguenza, lo spazio di azione degli attaccanti si riduce sensibilmente.

---

3. Lavoro svolto

L'obiettivo del lavoro svolto, e descritto in questa tesi, è di rendere più efficace il sistema Psyco-Virt, aggiungendo nuovi controlli ed estendendo le funzionalità di quelli gia presenti. Inoltre, è stata sviluppata una nuova libreria di introspezione di "alto livello", chiamata Xen-VMI (vedi sezione 3.2.3), che permette di interpretare i dati di "basso livello" in modo più semplice e strutturato rispetto alla precedente libreria di Psyco-Virt. La libreria Xen-VMI, comprende un insieme di funzioni che implementano dei controlli su un insieme di dati sensibili del kernel degli OS. I controlli in introspezione già presenti, che utilizzavano la libreria precedente, sono stati integrati nella libreria Xen-VMI.

Globalmente, il sistema è stato irrobustito ed è stato aumentato il grado di sicurezza delle macchine virtuali, in modo da aumentare il numero degli attacchi che si possono rilevare. La migliore capacità di rilevazione è dovuta anche agli importanti software di protezione installati sulle VM, come Grsecurity e LIDS (vedi sezione 3.2.4). Sono stati condotti dei test per valutare il grado di efficacia dei controlli sviluppati e dimostrare che la sicurezza delle macchine virtuali è effettivamente aumentata; questo tema sarà trattato nella sezione 4.1. Infine, sono stati condotti dei test per verificare l'efficienza delle nuove componenti, in termini di overhead introdotto nel sistema. I test, come vedremo nella sezione 4.2, dimostrano che il calo di performance è relativamente contenuto e quindi accettabile.

La complessità dei controlli, la cui implementazione sarà illustrata nella sezione 3.2, è minima: ciò è permesso dalla tecnologia della VMI che offre la possibilità di esaminare lo stato di un host dall'esterno con la stessa accuratezza di un controllo locale e con un alto livello di sicurezza.

3.1 Problemi affrontati

Questa sezione descrive le vulnerabilità delle parti dei sistemi che permettono gli attacchi più diffusi ed utilizzati. I problemi di sicurezza considerati riguardano:

- interfacce di rete
- moduli del kernel
- codice e strutture dati del kernel
- processi

3.1.1 Interfacce di rete, sniffing

Il problema più rilevante, che riguarda le interfacce di rete, è lo *sniffing* [16].

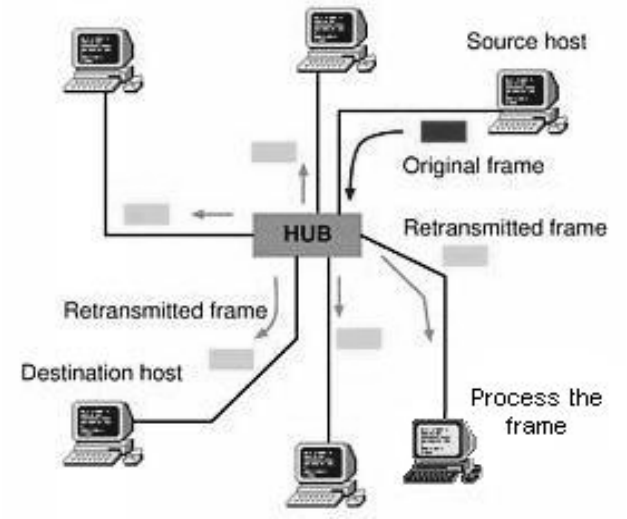


Fig.5: Packet sniffing

Lo sniffing è una tecnica che permette ad un utente di leggere nel dettaglio il traffico che transita nella rete, in ingresso e in uscita, in modalità passiva. Gli scopi possono essere legittimi, ad esempio per verificare se vi sono state eventuali intrusioni nel sistema, o illegittimi, ad esempio per intercettare informazioni sensibili provenienti da terze parti.

Uno *sniffer* locale, cioè il programma che implementa lo sniffing, imposta la scheda di rete di una macchina in *modalità promiscua*, comunicando al kernel la volontà di accettare tutti i pacchetti, anche quelli non indirizzati alla macchina considerata. Vi sono diversi modi per impostare la modalità promiscua, alcuni sono rilevati anche dal programma di sistema *ifconfig*. Altri che, ad esempio, utilizzano la chiamata di sistema *setsockopt()*, agiscono ad un livello più basso di quello del tool `ifconfig` e non sono rilevati in user space.

Riassumendo, per poter "sniffare" i pacchetti che attraversano la scheda di rete, uno sniffer deve impostare in modalità promiscua la corrispondente interfaccia di rete dell'host, andando a modificare gli opportuni parametri (*flags*). Questa modifica, diretta o indiretta, viene registrata nel kernel della macchina.

3.1.2 Moduli del kernel (LKM)

I moduli del kernel [22] sono componenti software che permettono di estendere il kernel, senzainterromperne l'esecuzione. In questo modo, è possibile ampliare le funzionalità del kernel senza dover riavviare il sistema o ricompilare il kernel. I moduli del kernel sono entità realizzate per essere utilizzate su kernel *monolitici* e Linux ne fa un uso massiccio.

Un *Loadable Kernel Module (LKM)* può essere usato in molti modi, tutti molto importanti nei sistemi moderni. Essi permettono di implementare driver per dispositivi o per i filesystem, di definire nuovi protocolli di rete e nuove system call, ed anche di ridefinire quelle già esistenti, e creare interpreti per eseguibili. Tutto questo viene svolto senza dover ricompilare il kernel anzi, è possibile eseguire codice in spazio kernel solo quando effettivamente necessario, inserendo e rimuovendo dinamicamente i LKM durante l'esecuzione del kernel. Questa versatilità è un grande vantaggio per l'amministratore del sistema, ma anche per chiunque voglia prenderne il controllo in maniera illegittima. Infatti, i LKM sono uno strumento molto utilizzato dagli attaccanti per eseguire i propri rootkit in "kernel space". Uno tra i tanti attacchi possibili, e tra i più comuni, effettua la redirezione di una o più chiamate di sistema modificando la *System Call Table* [11]. Questo consente di eseguire codice maligno quando un qualsiasi processo in esecuzione invoca una delle *System Call* target dell'attacco. Inoltre, un attacco del genere permette all'attaccante di nascondere molto bene le sue tracce.

3.1.3 Codice e strutture dati del kernel

Molte parti del kernel sono particolarmente sensibili agli attacchi, ed è importante garantire la loro integrità perché, se fossero compromesse, si avrebbero serie ripercussioni sulla stabilità del sistema.

Le parti del kernel da noi prese in considerazione sono quelle che sono compromesse dai kernel-rootkit più diffusi e conosciuti [11,24,40]:

- Codice delle System Call
- Codice degli handler per la gestione delle interruzioni/eccezioni
- Tabella delle System Call
- Tabella delle interruzioni
- Virtual File System

Il codice delle *System Call* e degli *handler* delle interruzioni/eccezioni viene modificato da molti kernel-rootkit. In questo modo, le procedure legali sono rimpiazzate da quelle degli attaccanti, che riescono ad eseguire codice arbitrario nel kernel. Il codice eseguibile di queste procedure si trova nella *text area* del kernel, che contiene il codice statico di molte altre funzionalità del sistema. Le tabelle delle System Call e delle interruzioni contengono, rispettivamente, i puntatori al codice delle System Call e al codice degli handler per gestire, ad esempio, le interruzioni di I/O e le eccezioni di *Page Fault*. Il *Virtual File System* (VFS) contiene le strutture dati utilizzate da programmi di sistema, come *ps*, per accedere alle informazioni dal file system `/proc`.

Tutte queste parti del kernel, solitamente, sono "read-only", quindi non devono essere modificate durante l'esecuzione del kernel. Si rilevano delle modifiche solo quando un attaccante ha cambiato il valore di un puntatore o una parte del codice delle procedure, per poter eseguire codice maligno in spazio kernel e nascondere la sua presenza.

3.1.4 Processi, buffer owerflow

Nei sistemi operativi Linux il kernel ha il compito di gestire lo spazio di indirizzamento dei processi [27]. Attraverso il kernel, un processo può dinamicamente aggiungere e rimuovere aree di memoria al proprio spazio di indirizzamento, o memoria virtuale.

Un'area di memoria può contenere, ad esempio:

- il codice eseguibile del programma (*text section*)
- le variabili globali inizializzate del programma (*data section*)
- spazio libero per le variabili globali non inizializzate (*bss section*)
- lo *stack* di esecuzione in spazio utente, distinto dallo stack del kernel
- spazio supplementare per le sezioni text, data e bss delle librerie utilizzate, ad esempio quelle del C
- tutti i file mappati

Il *memory descriptor* contiene tutte le informazioni relative alla memoria virtuale di un processo, cioè su tutte le aree di memoria che il processo che può indirizzare. Ad ogni area di memoria sono associati dei permessi, ad esempio: lettura, scrittura, esecuzione,

che limitano gli accessi per motivi di sicurezza. Ad esempio, nello spazio riservato al codice eseguibile non è possibile effettuare scritture, pena la terminazione del processo.

I problemi di sicurezza, in effetti, non sono dovuti alla text area, che contiene dati inizializzati e statici, ma alle aree per i dati non inizializzati, che sono più difficili da controllare. Queste aree sono soggette ad attacchi di tipo *buffer overflow* [28], che rappresentano la stragrande maggioranza di tutti gli attacchi ai sistemi informatici.

Il buffer overflow consiste nello scrivere oltre il limite di un buffer, un blocco contiguo di memoria che contiene più istanze dello stesso tipo di dato, e sovrascrivere alcune informazioni relative al flusso di esecuzione di un programma. Attraverso un attacco di tipo buffer overflow, un attaccante può modificare parte dello stato interno del programma affetto da alcune vulnerabilità [26]. Ad esempio, la maggior parte dei programmi è scritta con il linguaggio C, che non esegue, automaticamente, nessun controllo sulla dimensione delle variabili non inizializzate. Questo può consentire di eseguire codice arbitrario e, nel caso in cui il software abbia sufficienti privilegi, è possibile impadronirsi dell'host, ottenendo una shell di root e controllandolo da remoto. Fortunatamente, questo non è sempre possibile perché chi attacca il sistema deve preparare il codice da eseguire nello spazio d'indirizzamento del programma, individuare una posizione adeguata e permettere all'applicativo di saltare a quella porzione di codice con parametri esatti, caricati nei registri e nella memoria.

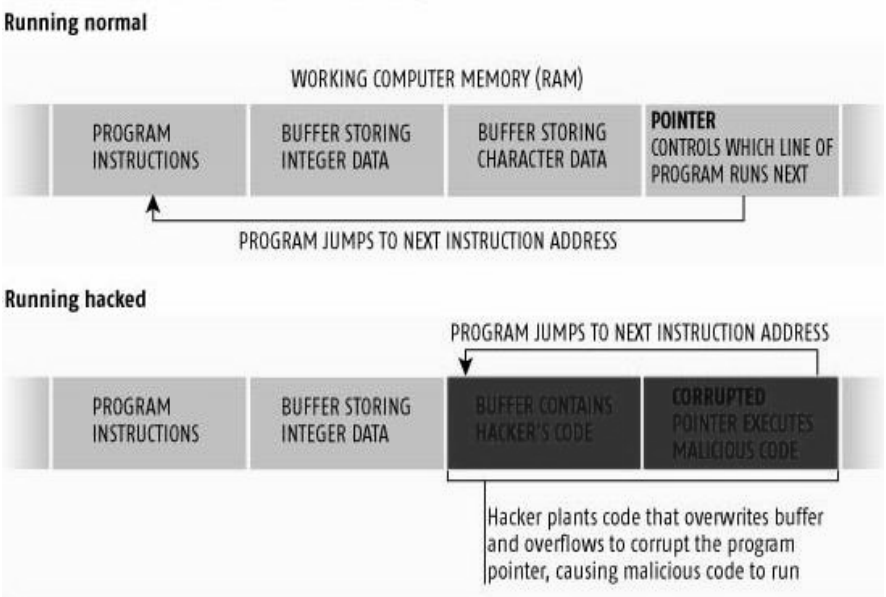


Fig.6: Buffer overflow

La tecnica più utilizzata per eseguire un *exploit* basato su buffer overflow è mostrata nella Fig.6. Essa consiste nell'individuare una variabile automatica soggetta al possibile overflow e inviare al programma una stringa molto grande in modo da riempire oltre il limite il buffer, per poter cambiare il record di attivazione della procedura. In questo modo, l'iniezione del codice maligno e la corruzione del buffer non avvengono contemporaneamente. L'attaccante può iniettare il codice in un buffer senza provocare un overflow e riempire oltre il limite un altro buffer, corrompendo il puntatore di quest'ultimo per farlo puntare al codice maligno. Il buffer oggetto dell'overflow non deve avere controlli sulla sua dimensione, altrimenti sarà possibile fare un overflow solo di pochi byte. In questo caso, il codice risiederà in un buffer sufficientemente grande e che sarà richiamato dal buffer in overflow.

3.2 Implementazione dei controlli

I controlli possono essere suddivisi in due categorie:

- Quelli che analizzano i valori dei dati ottenuti mediante introspezione, controllando che siano uguali a quei valori che indicano uno stato corretto per il sistema.
- Quelli che confrontano i dati ottenuti mediante introspezione con quelli forniti dal sistema monitorato, per scoprire eventuali differenze tra i due dati. I dati dal sistema sono ottenuti invocando una funzione ad un livello più alto rispetto all'introspezione.

Tutti i controlli implementati sono temporizzati, dunque, una volta attivati, sono eseguiti ad intervalli di tempo regolare e possono essere interrotti e riavviati in qualsiasi momento, senza dover riattivare il sistema del dominio privilegiato o quelli dei domini guest.

I controlli sono stati implementati in due modi, che garantiscono lo stesso livello di sicurezza:

- Controlli eseguiti sui dati ottenuti tramite introspezione e raggruppati tutti nella libreria di alto livello *Xen-VMI* (vedi sezione 3.2.3);
- Controlli eseguiti sui dati forniti del sistema, a livello utente o a livello kernel, dagli agenti in esecuzione sulle VM monitorate.

Nel secondo caso, bisogna verificare la corretta esecuzione dell'agent: ciò può avvenire in uno dei seguenti modi:

- Controllando che l'agent non sia manomesso, ad esempio controllando le pagine di memoria relative alla text area del processo associato;
- Controllando che l'output prodotto dall'agent sia uguale all'output prodotto da un controllo dello stesso tipo ma implementato come una funzione di introspezione.

È importante notare quello che è stato detto nella sezione 2.2, a proposito dei due step che garantiscono la sicurezza del sistema. Il *Trusted Computing Base* del sistema include il processo di introspezione, che verifica: (1) l'integrità delle parti critiche del nucleo (kernel) dei sistemi monitorati e (2) la corretta esecuzione degli agent, eseguiti sui sistemi monitorati. Per questo motivo, anche gli agent fanno parte del TCB. In questo modo si sfruttano a pieno, e nel modo più efficiente, i vantaggi introdotti dall'uso del livello della virtualizzazione e quelli offerti dalle tecniche per la rilevazione di intrusioni. Infatti, la soluzione adottata minimizza la complessità del processo di introspezione e dell'apposita libreria che fornisce la necessaria visione di alto livello per elaborare i dati, poiché essa deve "ricostruire" solo le strutture dati del kernel. Gli agenti e le altre funzioni di sicurezza (vedi sezione 3.2.4), installate sulle VM monitorate, verificano la consistenza di altre parti degli OS e dei processi a livello utente. Questa architettura di sicurezza consente, in molti casi, di scegliere come implementare un controllo. Ad esempio, se dobbiamo controllare una grossa quantità di dati, conviene implementare un agent, viceversa è più efficiente implementare una funzione di introspezione. Come descritto nella sezione 4.2, abbiamo implementato uno dei nostri controlli in entrambi i modi, allo scopo di verificare meglio le caratteristiche delle due implementazioni.

3.2.1 Controllo sulla modalità promiscua

Questo controllo permette di rilevare se un'interfaccia di rete del sistema monitorato sta operando in modalità promiscua. Quando viene rilevata questa situazione, è molto probabile che ci sia uno sniffer attivo sul sistema e, in ogni caso, è una situazione non permessa.

Il controllo è stato implementato definendo il modulo (vedi sezione 3.1.2) *antisniff*, un programma scritto in linguaggio C nel quale sono incluse ed utilizzate le librerie definite per implementare un LKM [36] (*header* `<linux/module.h>`).

La struttura del kernel, che contiene le informazioni sui *network device* (*struct net_device*) è definita in `<linux/netdevice.h>`. Il modulo viene "caricato" sul dominio da monitorare ed analizza, ad intervalli di tempo regolare, i campi *flags* e *promiscuity* delle strutture che contengono informazioni sui device *"eth"*, che è il suffisso del nome di tutte le interfacce di rete. La lista dei dispositivi di rete è memorizzata a partire dall'indirizzo *dev_base*. Se nel campo flags è settato il falg *IFF_PROMISC*, o se il campo promiscuity ha valore 1, l'interfaccia di rete corrispondente è in modalità promiscua. Quando si rileva che un interfaccia di rete è in modalità promiscua, si deve bloccare l'esecuzione del dominio e poi scoprire la causa di questa situazione, in particolare individuare lo sniffer.

L'integrità del controllo, e in particolare la corretta esecuzione del modulo `antisniff`, è garantita dal controllo sui moduli del kernel, descritto nella sezione 3.2.3.2.

3.2.2 Controllo dell'integrità dei processi

Questo tipo di controllo ha lo scopo di verificare che nessun processo eseguibile sull'host monitorato sia modificato, ovvero sostituito da una versione modificata dall'attaccante. Il controllo deve essere periodico, e deve essere applicato su quelle informazioni dei processi che sono significative per verificarne l'autenticità. La *text area* di un processo è il target ideale per questo controllo, perché contiene le istruzioni eseguibili, che identificano in modo completo un processo.

La procedura di controllo prevede di calcolare, in un momento sicuro per il sistema, l'*hash* della text area dei processi eseguibili sull'host monitorato, e ricalcolarlo ad intervalli di tempo regolare. Ogni cambiamento nel valore dell'hash segnala una modifica. L'esecuzione di questo tipo di controllo richiede di conoscere gli hash di tutti i processi che possono essere eseguiti sul dominio, per questo motivo occorre stabilire in anticipo tutti e soli i processi eseguibili sull'host monitorato. Idealmente, il fornitore del software dovrebbe allegare alla release l'hash del codice, che dovrebbe essere aggiunto automaticamente al database degli hash dei programmi eseguibili sul sistema. Attualmente, gli hash sono calcolati ed inseriti nel database manualmente. Prima di attivare il controllo, vengono attivati tutti i processi consentiti, e viene creata la cosiddetta lista di controllo, che associa al nome di ogni processo l'hash della sua text area. Questo procedimento è vulnerabile, perché non verifica se, al momento dell'attivazione del

controllo, i processi in esecuzione sono quelli autentici; in questo modo, esso rischia di legittimare dei malware.

Per ragioni di efficienza, il controllo è stato implementato in un modulo del kernel (vedi sezione 3.1.2), chiamato *check_proc*. Per calcolare l'hash della text area di ogni processo, è stato scelto l'algoritmo *SHA1*. Il kernel rappresenta i processi con strutture di tipo *task_struct* definite in `<linux/sched.h>`. Ogni "task" ha accesso al *memory descriptor* tramite il corrispondente campo *mm*. Esso contiene tutti gli indirizzi a cui sono allocate tutte le strutture relative ai processi, in particolare gli indirizzi virtuali di inizio e fine text section.

Tramite la funzione *access_process_vm()*, la funzione *ptrace_kreaddata()*, implementata modificando la funzione *ptrace_readdata()* di Linux, riloca gli indirizzi virtuali ed accede all'indirizzo *start_code* del processo copiando l'intera text area. La dimensione dell'area viene ottenuta calcolando la differenza tra l'indirizzo finale, *end_code*, e l'indirizzo iniziale, *start_code*.

```
int ptrace_kreaddata(struct task_struct *tsk,
                  unsigned long src, char __user *dst, int len)
{
     int copied = 0;

     while(len > 0)
     {
            char buf[128];
            int this_len, retval;
            this_len = (len > sizeof(buf)) ? sizeof(buf) : len;
            retval = access_process_vm(tsk, src, buf, this_len, 0);
            if(!retval) {
                  if(copied) break;
                  return -EIO;
            }
            if(memcpy(dst, buf, retval) == NULL)
                  return -EFAULT;

            copied += retval;
            src += retval;
            dst += retval;
            len -= retval;
     }

     return copied;
}
```

Fig.7: Codice della funzione `ptrace_kreaddata()`

Per implementare il calcolo dell'hash sono state utilizzate le funzioni *crypto_digest* di Linux, che forniscono un'interfaccia per l'uso dell'algoritmo SHA1.

Al momento della verifica, per ogni task nella “task list”, il modulo esegue il procedimento appena descritto, che prevede: (1) recupero della text area, (2) calcolo dell'hash e (3) verifica del valore sulla lista di controllo contenente i digest corretti. È da considerare illecito sia il caso in cui si rileva la presenza di un processo non nella lista dei processi eseguibili, e quindi senza nessun hash associato, sia il caso in cui il digest sia diverso dal corrispettivo nella lista di controllo. In questi casi, deve essere sospesa l'esecuzione del dominio monitorato e devono essere effettuate ulteriori verifiche.

3.2.3 Libreria di introspezione (Xen-VMI)

Questa libreria contiene tutte le funzioni che accedono ai dati del kernel in introspezione. Queste funzioni sono state implementate con il linguaggio C e sono state compilate in modo da includere gli *header del kernel* relativi agli OS monitorati. Gli header del kernel sono lo strumento che consente di colmare il “gap semantico” tra quella che è la visione del sistema dall'interno e quella ottenuta attraverso l’introspezione.

Lo sviluppo di ogni funzione può essere suddiviso in tre fasi:

- individuazione dei tipi di dato delle informazioni che si vogliono controllare;
- introspezione delle specifiche informazioni;
- controllo dei dati ottenuti nella fase precedente.

Gli header del kernel sono lo strumento utilizzato nella prima fase e contengono la descrizione di tutti i tipi di sttrutture dati che memorizzano le informazioni utilizzate dal kernel. I controlli locali, implementati come moduli del kernel (vedi sezione 3.1.2), e descritti nelle sezioni 3.2.1 e 3.2.2, sono installati ed eseguiti nel kernel del dominio monitorato ed hanno accesso diretto alle strutture dati del kernel, dichiarate dai relativi header. Il manager, invece, è un processo eseguito in user space e non può accedere direttamente alle strutture dati del kernel delle VM che, tra l’altro, è diverso da quello del dominio 0 che esegue il manager. La soluzione utilizzabile è quella che prevede di copiare la zona di memoria che contiene i dati che interessano nello spazio di indirizzamento del processo associato al manager. Il manager è stato compilato in modo da includere gli header del kernel delle VM monitorate, in questo modo esso può ricostruire le strutture dati che descrivono il contenuto della memoria a livello fisico, evitando di doverle ridefinire “manualmente”. Solitamente, le strutture sono molto

complesse e variano da una versione all'altra del kernel, quindi è fondamentale interfacciarsi con gli header del kernel.

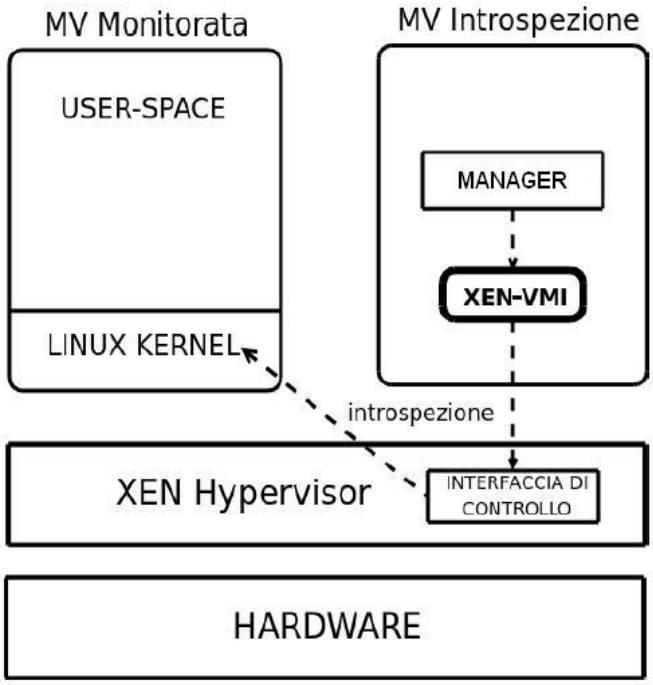


Fig.8: Schema della libreria Xen-VMI

La precedente libreria di introspezione di "alto livello", utilizzata da Psyco-Virt, era *XenAccess* [35,48]. Questa libreria, fornisce un insieme di funzioni che fungono da interfaccia verso la libreria di controllo di Xen e consentono di effettuare in modo più semplice il processo di introspezione. Tuttavia, XenAccess fornisce una visione di "alto livello" ricostruendo manualmente le strutture dati del kernel; ciò richiede di utilizzare degli offset di valore statico per scorrere la memoria fisica e separare i campi delle strutture. Xen-VMI utilizza le funzioni di interfaccia di XenAccess, ma ricostruisce le strutture dati del kernel, utilizzando le dichiarazioni contenute negli header del kernel. Inoltre, utilizza le funzioni della libreria di controllo di Xen per agire sullo stato di esecuzione delle VM, ad esempio, per bloccare l'esecuzione di un dominio.

Le fasi dettagliate del processo di introspezione, implementato da ogni funzione della libreria Xen-VMI, sono le seguenti:

1) Si sospende l'esecuzione del dominio monitorato.
2) Si ricava l'indirizzo base relativo alla struttura dati a cui sono applicati i controlli. Il file *System.map* nella directory `/boot`, elenca le corrispondenze tra gli

indirizzi di tutte le strutture dati del kernel eseguito sulle VM monitorate e l'alias dell'indirizzo, detto *kernel symbol*.

3) Vengono mappate nello spazio di indirizzamento del manager tutte le pagine di memoria del dominio relative alle strutture dati del kernel di interesse, a partire dall'indirizzo ricavato al punto 2.

4) Viene effettuato il "casting" della memoria fisica alle strutture dati corrispondenti a quelle utilizzate dal kernel. Come gia detto, questo è possibile perché la dichiarazione delle strutture dati è estratta dagli header del kernel di Linux.

5) A questo punto, è stata definita la semantica dei dati associandogli un tipo ed essi possono essere controllati dal manager. In molti casi, il controllo prevede di calcolare l'hash delle informazioni e poi confrontare il valore ottenuto con quello registrato in un momento ritenuto sicuro per lo stato del sistema. L'algoritmo usato per calcolare l'hash dei dati è *SHA1*, che è invocato interfacciandosi con le funzioni della libreria *OpenSSL* [29].

6) Viene riavviata l'esecuzione del dominio monitorato.

Per illustrare meglio questi passi, consideriamo, ad esempio, l'implementazione della funzione di introspezione che ricostruisce la lista dei processi in esecuzione su un dominio monitorato ed illustrata in Fig.9.

```
int get_process_list(uint32_t dom)
{
     xa_instance_t xai;
     unsigned char *memory = NULL;
     uint32_t offset, init_addr, next_addr;
     struct task_struct *task;
     char *symbol = "init_task";


     if(xa_init(dom, &xai) == XA_FAILURE) goto error_exit;

1.   xc_domain_pause(xai.xc_handle, dom);

2/3. memory = xa_access_kernel_symbol(&xai, symbol, &offset);
     if(NULL == memory) goto error_exit;

     if(linux_system_map_symbol_to_address(
            &xai, symbol, &init_addr) == XA_FAILURE) goto error_exit;

     while(next_addr != init_addr)
     {
4.      task = (struct task_struct *)(memory + offset);
        next_addr = (uint32_t)next_task(task);
5.      /*do something whit task */
```

```
        munmap(memory, XA_PAGE_SIZE);
3.      memory = xa_access_virtual_address(&xai, next_addr, &offset);
     }
6.   xc_domain_unpause(xai.xc_handle, dom);

error_exit:
     xa_destroy(&xai);
     if(memory) munmap(memory, XA_PAGE_SIZE);

     return 0;
}
```

Fig.9: Codice di una funzione di introspezione

Le parti del sistema che vengono analizzate in introspezione sono:

- Dispositivi di rete
- Moduli del kernel (LKM)
- Tabella delle system call
- Tabella delle interruzioni (IDT)
- Text area del kernel
- Lista dei processi
- Lista dei file aperti

3.2.3.1 Dispositivi di rete

L'introspezione sulle interfacce di rete deve implementare lo stesso controllo del modulo `antisniff`. Quindi, le informazioni restituite dalla funzione sono le stesse descritte nella sezione 3.2.1.

L'approccio per definire questo controllo è simile a quello implementato nel tool *Kstat* [10]. Mentre però Kstat accede ai dati dallo spazio utente, tramite il file speciale `/dev/kmem`, la nostra funzione di introspezione, *if_promisc()*, preleva i dati dal kernel eseguendo l'introspezione dal livello del VMM, che non può essere compromesso.

Entrambe le versioni del controllo, quella del modulo `antisniff` e quella della funzione di introspezione, offrono lo stesso livello di sicurezza, ma l'implementazione tramite introspezione è globalmente più rigorosa dal punto di vista dell'indipendenza del controllo dal sistema.

3.2.3.2 LKM

Una caratteristica importante ed utile dei LKM è la loro dinamicità. Questo però ne complica il controllo. Per questo motivo, è stato necessario imporre un vincolo importante: si devono conoscere in anticipo tutti i moduli del kernel eseguibili sul sistema. In questo modo, un modulo non appartenente alla lista di quelli conosciuti ed autorizzati inizialmente è considerato non accettabile ed è segnalato come problema.

Il simbolo *modules* rappresenta l'alias dell'indirizzo base della lista delle strutture di tipo *module* e definite in `<linux/module.h>`. I moduli sono concatenati in una lista bidirezionale circolare. Per ogni modulo occorre restituire il nome, la dimensione, l'indicatore della licenza e la sezione contenente il codice eseguibile del modulo (*core_text_section*).

Il nome del modulo è chiaramente il primo parametro da controllare per verificare se esso è presente in quella che è la lista dei moduli permessi. Ogni modulo del kernel dovrebbe avere una licenza *GPL* o derivata: introdurre nel kernel moduli senza licenza è molto pericoloso. Infatti, alcuni kernel non permettono neanche l'esecuzione di un modulo senza licenza. Il vero controllo viene però effettuato sulla text area dei LKM. Infatti, per un attaccante che riesce ad ottenere un accesso root al sistema non è difficile eseguire un proprio modulo GPL compatibile e con lo stesso nome di un modulo regolare. Per questo motivo, occorre verificare che i moduli non siano "sovrascritti". Questo viene fatto calcolando l'hash del codice di ogni modulo mediante questa procedura:

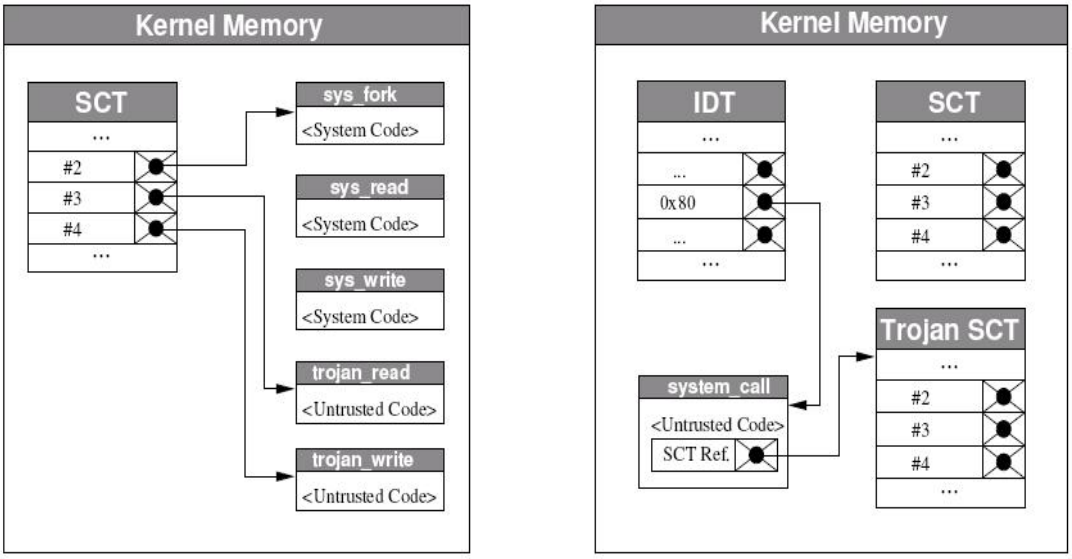
1) prima di attivare il controllo, si caricano nel kernel tutti i moduli consentiti;
2) si attiva il controllo, cioè si esegue il manager che invoca per la prima volta la funzione di introspezione *get_modules_infos()*. La funzione costruisce la lista di controllo, associando al nome di ogni modulo l'hash della sua text area. I valori registrati in questa fase sono considerati autentici e corretti;
3) ad intervalli di tempo regolare, il manager invoca la funzione che calcola la lista degli hash del codice dei moduli rilevati nel momento in cui è eseguita. A questo punto, il manager confronta questa lista con quella di controllo, per verificare se l'hash corrispondente ad ogni modulo è quello corretto e se sono presenti moduli non consentiti. Nei casi irregolari viene bloccata l'esecuzione del dominio monitorato.

Ci sono due aspetti da notare. Dal punto di vista della sicurezza, calcolare l'hash della text area dei moduli non ha nessuna utilità, è semplicemente un modo pratico e veloce, per implementare il confronto tra il contenuto di zone di memoria relativamente grandi. Inoltre, se l'hash fosse fornito dagli sviluppatori, l'attaccante non avrebbe la possibilità di eludere il controllo, modificando uno o più moduli prima che l'amministratore attivi il controllo. Comunque, quest'ultimo aspetto, pur essendo rilevante dal punto di vista della sicurezza, non è strettamente legato alle tematiche da noi affrontate.

3.2.3.3 System Call Table

La System Call Table (SCT) è la tabella che contiene gli indirizzi base di tutte le chiamate di sistema. Ogni System Call (SC) ha associato un puntatore all'indirizzo, a partire dal quale è memorizzato il codice della funzione. La lista dei puntatori si trova all'indirizzo associato al "kernel symbol" *sys_call_table* e non deve essere modificata, visto che il posizionamento di ogni SC nel kernel è fisso e statico.

Il controllo ha l'obiettivo di scoprire modifiche al valore dei puntatori alle SC, che indicherebbero una *redirezione* (vedi Fig.10a), sicuramente opera di un attaccante.



(a) (b)

Fig.10: Redirezione delle System Call

La soluzione adottata per realizzare questo controllo è quella di calcolare l'hash del contenuto della tabella in un momento sicuro per il sistema e verificare, ogni volta che il

manager applica l'introspezione, invocando la funzione *get_sys_call_table()*, che il valore dell'hash non cambi. Se vengono rilevate discrepanze tra l'hash calcolato in un certo momento e l'hash corretto, si blocca il dominio su cui si è rilevata l'anomalia, in attesa di ulteriori verifiche.

3.2.3.4 Interrupt Descriptor Table

L'Interrupt Descriptor Table (IDT), memorizzata all'indirizzo *idt_table*, è la tabella che contiene i puntatori all'indirizzo base di ogni handler definito per gestire:

- le interruzioni di I/O,
- le eccezioni, ad esempio fault di pagina e invocazione di SC.

In realtà, i controlli di consistenza sono applicati sulla *trap_table* [44], cioè la tabella dei descrittori delle interruzioni gestita da Xen. I tipi di attacchi considerati, in primis la redirezione (vedi Fig.10b), e la soluzione adottata per verificare l'integrità delle informazioni sono equivalenti al caso descritto nella sezione precedente, relativo alla System Call Table. La funzione che implementa l'introspezione è *get_idt_table()*.

3.2.3.5 Kernel Text Area

La text area del kernel è quella compresa tra l'indirizzo *_text* e l'indirizzo *_etext* del kernel. Questa sezione del nucleo contiene il codice statico delle funzioni del kernel, in particolare quello:

- di tutte le system call;
- degli handler che gestiscono le eccezioni e le interruzioni di I/O.

Il controllo implementato ha l'obiettivo di rilevare il cosiddetto *stealth* delle SC o degli handler, che consente agli attaccanti di eseguire codice maligno nello spazio kernel, modificando il codice eseguibile delle procedure (vedi Fig.11).

Anche in questo caso, la consistenza è controllata calcolando l'hash delle informazioni, in particolare, dell'intera text area del kernel. Qualsiasi tentativo di "stealth" nell'area è rilevato e classificato come un'intrusione. La funzione di introspezione è *get_kernel_text()*.

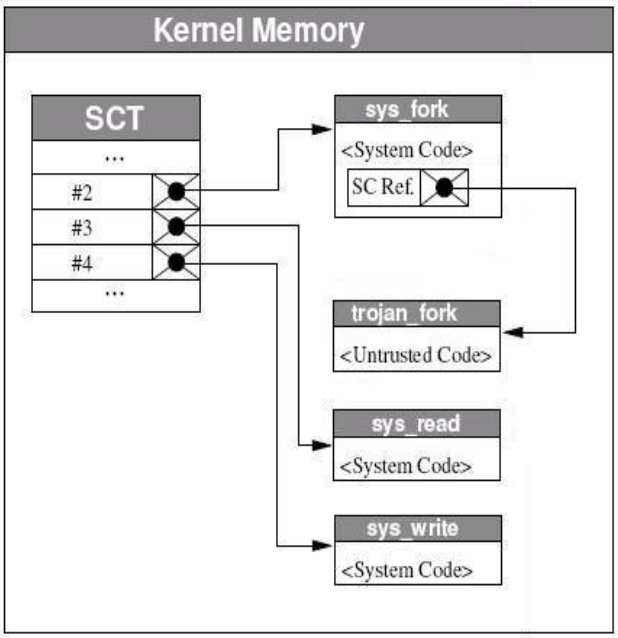


Fig.11: Esempio di stealth di una SC

3.2.3.6 Lista dei processi attivi

L'obiettivo del controllo sulla lista dei processi attivi (*running process list*) è quello di rilevare la presenza di processi nascosti sul sistema, cioè visibili ad un certo livello del sistema ma, invisibili ad un livello più alto. In particolare, il controllo di consistenza da noi implementato verifica che i dati forniti dal livello utente del dominio monitorato siano uguali a quelli restituiti dall'apposita funzione di introspezione *get_process_list()*.

All'indirizzo *init_task* è memorizzato il puntatore alla lista dei processi attivi. Le informazioni relative ad ogni processo, quali pid, user id, nome e stato, sono memorizzate nel kernel in una struttura di tipo *task_struct*, definita in `<linux/sched.h>`.

La funzione di introspezione restituisce in un formato prefissato le informazioni più comuni ed utili da verificare, che sono:

- identificatore (*pid*)
- user id (*uid*)
- user id effettivo (*euid*)
- nome (*comm*)

Il manager invoca la funzione e, contemporaneamente, invia la richiesta di esecuzione del comando *ps* all'agent in ascolto sull'host monitorato. L'agent esegue il comando *ps* ottenendo la lista dei processi attivi, visibile dal sistema in user space. La lista viene inviata al manager nello stesso formato adottato per la funzione di introspezione. A questo punto, il manager confronta i due output e segnala come processi nascosti tutti quei processi presenti nella lista ottenuta in introspezione, ma non nella lista restituita dall'agent. Inoltre, sono segnalate anche eventuali differenze nelle informazioni associate al processo, per esempio l'*uid* o l'*euid*.

3.2.3.7 Lista dei file aperti

Il controllo sulla lista dei file aperti da un processo è un ulteriore verifica delle informazioni riguardanti i processi. L'obiettivo del controllo è quello di esaminare la lista di tutti i file regolari aperti dai processi attivi sul sistema, per scoprire se qualche file è stato nascosto.

La struttura e la modalità del controllo (*agent-manger*) è la stessa di quello sulla "running process list" (vedi sezione 3.2.3.6). La lista dei file regolari aperti dai processi, calcolata mediante introspezione, viene confrontata con quella fornita, su richiesta dall'host monitorato e prodotta dall'apposito comando di sistema *lsof*.

La funzione di introspezione *list_open_files()* restituisce la lista dei file regolari aperti da ogni processo ed alcune informazioni di utilità per le verifiche successive. Le informazioni per ogni file sono:

- identificatore del processo (*pid*)
- identificatore del file (*inode*)
- nome del file, compreso il *path*

Di seguito, descriviamo le strutture dati del kernel utilizzate per reperire le informazioni sui file. Per ragioni di efficienza, le informazioni sui file sono sparse in molte strutture.

- *struct task_struct* in `<linux/sched.h>`: rappresenta i processi;
- *struct mm_struct* in `<linux/sched.h>`: rappresenta il "memory descriptor" di un processo;
- *struct vm_area_struct* in `<linux/mm.h>`: rappresenta un'area di memoria;

- *struct files_struct* in `<linux/file.h>`: rappresenta le informazioni sulla lista dei file aperti;
- *struct fdtable* in `<linux/file.h>`: rappresenta la lista dei file aperti da un processo;
- *struct file* in `<linux/fs.h>`: rappresenta i file;
- *struct inode* in `<linux/fs.h>`: rappresenta l'inode di un file;
- *struct dentry* in `<linux/dcache.h>`: rappresenta le informazioni sul path di un file.

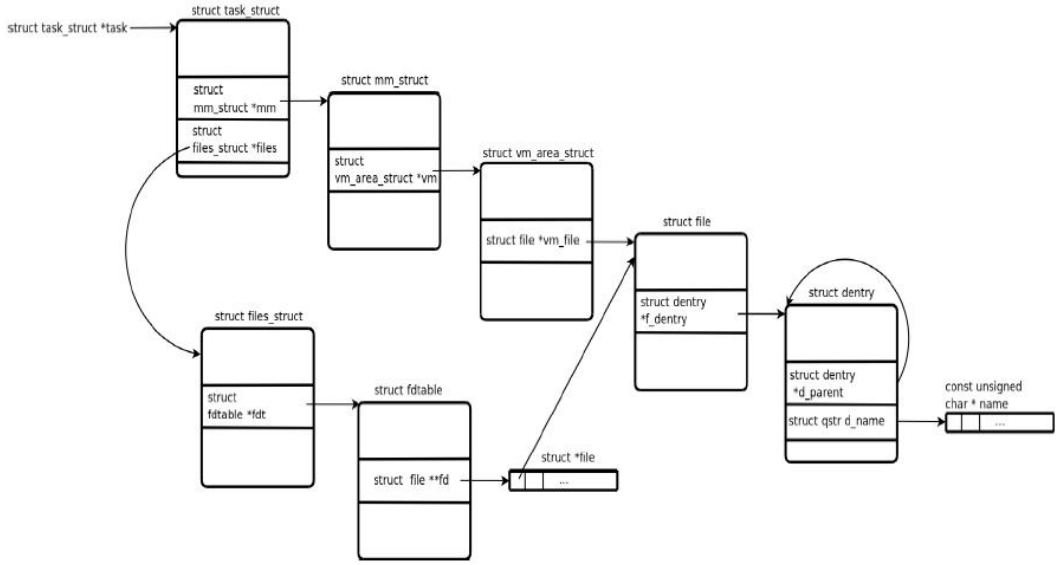


Fig.12: Schema della memoria riguardante i processi

La lista dei file aperti da un certo processo, con associato un relativo *file descriptor*, è puntata dal campo *task->files->fdt->fd*. Le informazioni relative ad un file mappato in una certa area di memoria di un processo sono reperibili in *task->mm->mmap->vm_file*.

Il campo *fd* è una lista di puntatori ai file descriptor, uno per ogni file aperto di un processo. Il descrittore del file contiene tutte le informazioni su un file aperto, come id, nome, dimensione e tipo. Le stesse informazioni, relative però ai file mappati, sono reperibili a partire da *task->mm->mmap->vm_file*. I file che sono mappati nello spazio di indirizzamento di un processo, possono essere, ad esempio, linker a librerie C.

Alcuni processi non possono avere dei file mappati sulla propria memoria, semplicemente perché non hanno e non utilizzano un proprio spazio di indirizzamento. I *kernel thread* (*kthread*) [27] sono eseguiti in "kernel space" ed utilizzano solo la memoria kernel, che è

unica e condivisa tra tutti i processi. Per questo motivo non hanno bisogno di un "memory descriptor", quindi, il campo *mm* della *task_struct* che li rappresenta ha valore *NULL*.

Le informazioni di utilità considerate, inode e path, sono memorizzate, rispettivamente, nei campi *f_dentry->d_name.name* e *f_dentry->d_inode->i_ino*. Il campo *f_dentry*, relativo alla struttura *dentry*, è un campo della struttura *file*, la stessa, sia per i file mappati, che per quelli aperti. Tramite la struttura *dentry* è possibile ottenere l'intero path del file.

I file a cui si applica questo controllo sono i file regolari. Essi sono individuati in base al valore del campo *f_dentry->d_inode->i_mode*, che indica il tipo del file, ad esempio regolare, directory, socket, device.

3.2.4 Altri controlli

Le macchine virtuali monitorate possono essere estese per includere un insieme di importanti controlli di sicurezza come quelli implementati, ad esempio, dai sistemi *Grsecurity* [12] e *LIDS* [25]. Questi sistemi sono in grado di rilevare e prevenire attacchi a più livelli del sistema ed aumentano la robustezza dell'OS stesso e quella dei processi a livello utente.

Grsecurity è un sistema che consiste in un insieme di patch per il kernel di Linux, che permettono di prevenire attacchi di tipo *buffer overflow*, come l'esecuzione di codice arbitrario nel kernel. Invece, LIDS (*Linux Intrusion Detection System*) è un insieme di patch che vengono applicate al kernel di Linux, per proteggere il file system di Linux da manomissioni, ad esempio sul file `/bin/login`, rilevare i *port scan* e gli *stealth scan*, ed implementare politiche MAC (*Mandatory Access Control*).

4. Test

In questa sezione descriviamo i test svolti per valutare l'efficacia dei controlli implementati, cioè la loro capacità di rilevare tentativi di intrusione e attacco, e la loro efficienza. L'efficienza è stata valutata in termini di overhead introdotto nel sistema.

I test sono stati eseguiti su un sistema con le seguenti caratteristiche:

- processore: AMD Athlon XP con frequenza di clock 1.833 GHz;
- dimensione memoria fisica: 512MB;
- immagine kernel distribuzione Fedora: 2.6.17-1.2187_FC5;
- immagine kernel dominio 0, distribuzione Fedora: 2.6.16-1.2133_FC5xen0.

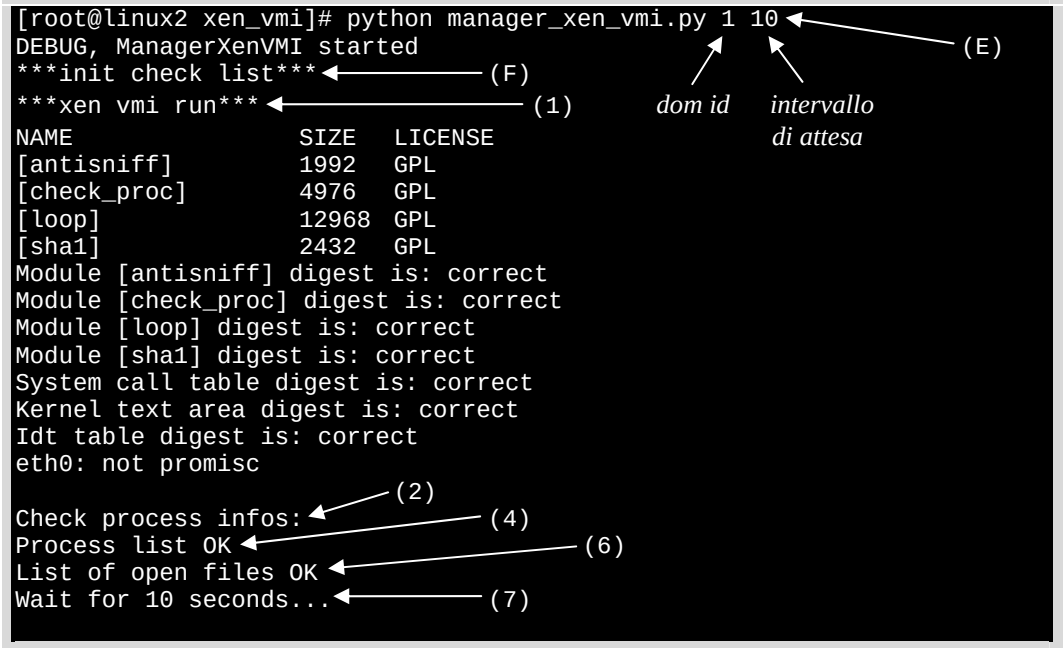
La macchina virtuale ha le seguenti caratteristiche:

- dimensione memoria: 64 MB;
- immagine kernel del domino utente, distribuzione Debian: vmlinuz-2.6.16-xenU (compilata dai sorgenti);

Il manager e gli agenti sono stati implementati con il linguaggio *Python* [34], che permette di usare in modo semplice strutture dati di alto livello, come stringe, liste e dizionari. Quando il manager viene attivato, viene specificato l'identificatore del dominio da monitorare e l'intervallo di tempo con cui effettuare i controlli periodici in introspezione. Questi controlli sono eseguiti dal programma C *xen_check_dom*. Il manager, prima attiva i controlli sui dati del kernel verificati, o calcolandone l'hash, o controllandone direttamente il valore e, successivamente, attiva i controlli sulla lista dei processi attivi e sulla lista dei file aperti. Questi controlli confrontantano i dati prodotti con quelli forniti dall'host monitorato. Oltre ai moduli `antisniff` e `check_proc`, l'host da controllare esegue due programmi, implementati anch'essi con Python, che fanno da "wrapper" verso i comandi di sistema *ps* e *lsof*. Questi programmi convertono l'output prodotto dai comandi in un formato che permette al manager di confrontarli con l'output prodotto, rispettivamente, dalle funzioni di introspezione *get_process_list()* e *list_open_files()*.

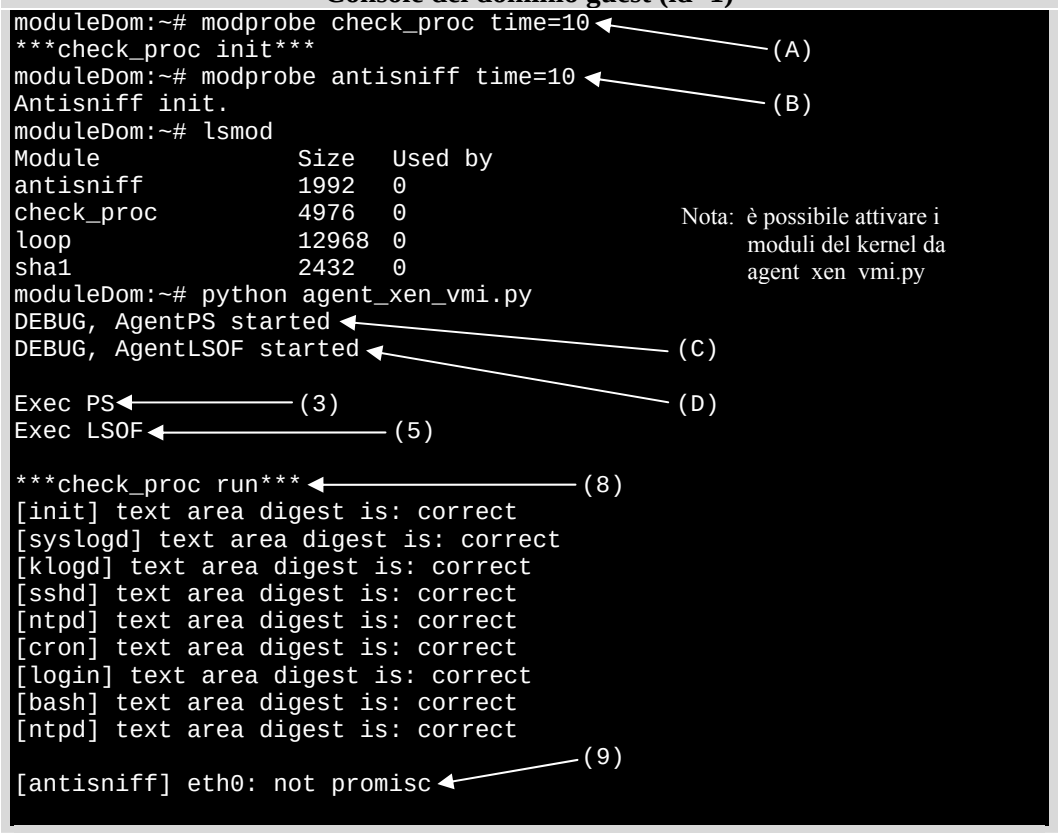
La Fig.13 illustra la procedura per attivare il manager e gli agenti (A-F) ed i messaggi che vengono stampati sulla console di amministrazione (Fig.13a) e su quella del dominio monitorato (Fig.13b). Con i numeri da 1 a 9 è indicata la cronologia delle fasi significative del procedimento di controllo che, nel caso considerato, sono ripetute ogni 10 secondi.

Console del dominio 0



(a)

Console del dominio guest (id=1)



(b)

Fig.13: Attivazione dei controlli

4.1 Efficacia

Per valutare l'efficacia dei nuovi controlli sono stati simulati diversi attacchi:

1) sono stati realizzati quattro rootkit per modificare (1) una entry nella tabella delle interruzioni, *trap_table*, (2) il puntatore nella tabella *sys_call_table*, (3) il codice di una *System Call* e (4) la *text area* di un processo in esecuzione.
2) è stato sostituito il modulo `antisniff` con una versione manomessa;
3) sono stati sostituiti i binari di sistema *ps* e *lsof* con versioni modificate per nascondere, rispettivamente, alcuni processi e file;
4) sono stati eseguiti diversi tipi di sniffer che settano la modalità promiscua dell'interfaccia di rete della VM monitorata in diversi modi [14].

Le precedenti componenti di Psyco-Virt non rilevavano, o rilevavano solo in parte, questi attacchi, che possono essere eseguiti da un attaccante remoto che ha ottenuto un accesso root al sistema. Per descrivere meglio i test, gli attacchi sono stati eseguiti sulla console di amministrazione del dominio monitorato, dove vengono visualizzate tutte le informazioni di controllo degli agenti.

Descrizione dei test

Il primo test ha lo scopo di verificare che le funzioni di Xen-VMI (1) *get_idt_table()*, (2) *get_syscall_table()* e (3) *get_kernel_text()*, rilevino le modifiche nell'area del kernel che controllano. Abbiamo simulato l'attacco sulle rispettive aree del kernel, implementando tre LKM che implementano le stesse funzionalità dei più noti rootkit kernel-level [15,23,38,40]. Inoltre, è stato implementato un programma per simulare un attacco sulla text area di un processo in esecuzione (4). Il programma utilizza la funzione `ptrace()` [41], che permette di modificare la memoria di un processo in esecuzione, ed alterarne il flusso di esecuzione. La Fig.14 descrive, come esempio, l'attacco al processo `cron`, con pid 1003, e la reazione dei controlli di consistenza, in seguito all'attivazione dei moduli maligni.

Console del dominio 0

```
...
***xen vmi run***
NAME                SIZE   LICENSE
[check_proc]        4976   GPL
[loop]              12968  GPL
[sha1]              2432   GPL
```

```
Module [check_proc] digest is: correct
Module [loop] digest is: correct
Module [sha1] digest is: correct
System call table digest is: correct
Kernel text area digest is: correct
Idt table digest is: correct
eth0: not promisc                          <-- intervallo di attesa (vedi Fig.12b)

***xen vmi run***                          <-- fine intervallo di attesa
NAME                SIZE   LICENSE
[syscall]           1892   GPL
[replace_idt]       1320   GPL
[replace_SC]        1768   GPL
[check_proc]        4976   GPL
[loop]              12968  GPL
[sha1]              2432   GPL
Module [syscall] digest is: NOT FOUND
Module [replace_idt] digest is: NOT FOUND
Module [replace_SC] digest is: NOT FOUND
Module [check_proc] digest is: correct
Module [loop] digest is: correct
Module [sha1] digest is: correct
System call table digest is: NOT correct
Kernel text area digest is: NOT correct
Idt table digest is: NOT correct
eth0: not promisc
```

(a)

Console del dominio guest

```
...
moduleDom:~# ***check_proc run***
[init] text area digest is: correct
[syslogd] text area digest is: correct
[klogd] text area digest is: correct
[sshd] text area digest is: correct
[ntpd] text area digest is: correct
[cron] text area digest is: correct
[login] text area digest is: correct
[bash] text area digest is: correct
[ntpd] text area digest is: correct
```

#il manager ha appena effettuato i controlli di consistenza

```
moduleDom:~# insmod replace_idt.ko          <-- (1)
[Redirezione di un handler]
- handler name: coprocessor_error
- handler address: 0xc01052a4
Quando il modulo sarà rimosso, verrà ripristinato il valore corretto
moduleDom:~# insmod syscall.ko              <-- (2)
[Redirezione di una System Call]
- System Call name: __NR_OPEN
- System Call address: 0xc015c390
Quando il modulo sarà rimosso, verrà ripristinato il valore corretto
moduleDom:~# insmod replace_SC.ko           <-- (3)
[Stealth di una System Call]
- System Call name: __NR_OPEN
- Stealth at address: 0xc015c390
Quando il modulo sarà rimosso, verrà ripristinato il valore corretto
moduleDom:~# ./stealth_proc 1003            <-- (4)
[Ptrace Injector]
- stealth process id: 1003
```

```
- stealth target: text area
```
#fine dell'intervallo di attesa (vedi Fig.12a)
```
moduleDom:~# ***check_proc run***
[init] text area digest is: correct
[syslogd] text area digest is: correct
[klogd] text area digest is: correct
[sshd] text area digest is: correct
[ntpd] text area digest is: correct
[cron] text area digest is: NOT correct
[login] text area digest is: correct
[bash] text area digest is: correct
[ntpd] text area digest is: correct
```

(b)

Fig.14: Test di efficacia 1

Il secondo test è stato condotto per dimostrare che un attaccante non può manomettere un agent del sistema, in particolare un agent implementato come modulo del kernel. L'attacco prevede che il modulo `antisniff` sia rimosso durante un intervallo di attesa di `xen_check_dom`, ed al suo posto sia caricata nel kernel una versione omonima ma che restituisce informazioni false. In particolare, essa segnala uno stato corretto anche quando l'interfaccia di rete è impostata in modalità promiscua. Nella Fig.15 viene mostrato e descritto questo attacco, in cui l'interfaccia di rete `eth0` viene simbolicamente settata in modalità promiscua, utilizzando il tool `ifconfig`.

Console del dominio 0

```
...
***xen vmi run***
NAME                SIZE   LICENSE
[antisniff]         1992   GPL
[check_proc]        4976   GPL
[loop]              12968  GPL
[sha1]              2432   GPL
Module [antisniff] digest is: correct
Module [check_proc] digest is: correct
Module [loop] digest is: correct
Module [sha1] digest is: correct
System call table digest is: correct
Kernel text area digest is: correct
Idt table digest is: correct
eth0: not promisc
```

intervallo di attesa (vedi Fig.13b)

```
***xen vmi run***
```

fine intervallo di attesa

```
NAME                SIZE   LICENSE
[antisniff]         1928   GPL
[check_proc]        4976   GPL
[loop]              12968  GPL
[sha1]              2432   GPL
Module [antisniff] digest is: NOT correct
Module [check_proc] digest is: correct
Module [loop] digest is: correct
Module [sha1] digest is: correct
System call table digest is: correct
Kernel text area digest is: correct
```

```
Idt table digest is: correct
eth0: PROMISC
```

(a)

Console del dominio guest

```
moduleDom:~# lsmod
Module              Size   Used by
antisniff           1992   0
check_proc          4976   0
loop                12968  0
sha1                2432   0
moduleDom:~# [antisniff] eth0: not promisc
```

#il manager ha appena effettuato i controlli di consistenza

```
moduleDom:~# ifconfig eth0 promisc
moduleDom:~# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:16:3E:2F:04:E0
          inet addr:10.1.0.3  Bcast:10.1.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7728 (7.5 KiB)  TX bytes:9333 (9.1 KiB)

moduleDom:~# modprobe -r antisniff
Antisniff exit.
moduleDom:~# insmod antisniff.ko
False Antisniff init.
```

#fine dell'intervallo di attesa (vedi Fig.13a)

```
moduleDom:~# [antisniff] eth0: not promisc
moduleDom:~# lsmod
Module              Size   Used by
Antisniff           1928   0
check_proc          4976   0
loop                12968  0
sha1                2432   0
moduleDom:~#
```

(b)

Fig.15: Test di efficacia 2

Il terzo test permette di valutare l'efficacia dei controlli sulla lista dei processi attivi e sulla lista dei file regolari aperti. Abbiamo simulato un attacco eseguendo il rootkit `tuxkit`, che nasconde alcuni processi in esecuzione, e abbiamo sostituito il binario *lsof* con una versione che filtra l'output dell'originale e nasconde alcuni file aperti. Queste modifiche alla lista dei processi attivi erano rilevate in modo simile dal sistema Psyco-Virt, i cui controlli utilizzavano una funzione della libreria XenAccess [48] per ottenere la lista degli identificatori dei processi in esecuzione. Utilizzando la funzione *get_process_list()* di Xen-VMI, è possibile ottenere, oltre al pid, altre importanti informazioni da controllare come, ad esempio, il livello di privilegio associato al processo. L'utilità di sistema *chkrootkit* [4] implementa dei controlli simili ai nostri perchè è in grado di rilevare modifiche ai comandi di sistema come *ls, ps, lsmod, find*.

Però, i controlli implementati dalle due funzioni di Xen-VMI, *get_process_list()* e *list_open_files()*, restituiscono delle informazioni più dettagliate e, soprattutto, questi controlli non sono eludibili dagli attaccanti.

Infine, il quarto test dimostra che i più comuni sniffer locali [14,42] sono rilevati sia dal modulo `antisniff` sia dal controllo che utilizza la funzione di introspezione *if_promisc()*. Gli sniffer impiegati per simulare l'attacco usano metodi diversi per settare la modalità promiscua, che solo in alcuni casi sono rilevati dalle utilità di sistema `ifconfig` e `chkrootkit`. Nella Fig.16 viene mostrato, come esempio, l'utilizzo dello sniffer *tcpdump* [42] da parte del dominio monitorato. Come si può vedere, il comando `ifconfig` non rileva la modalità promiscua, mentre l'agent `antisniff` rileva l'irregolarità.

Console del dominio guest

```
moduleDom:~# tcpdump eth0 &
moduleDom:~# [antisniff] eth0: PROMISC
moduleDom:~# ifconfig eth0
eth0      Link encap:Ethernet  Hwaddr 00:16:3E:2F:04:E0
          inet addr:10.1.0.3  Bcast:10.1.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7728 (7.5 KiB)  TX bytes:9333 (9.1 KiB)
```

Fig.16: Test di efficacia 4

4.2 Efficienza

Per quantificare l'overhead introdotto dai controlli di consistenza è stato eseguito il benchmark IOzone Filesystem [18], concorrentemente all'esecuzione dei nuovi controlli. Il manager esegue tutti i controlli di consistenza del kernel tramite l'utilizzo della libreria Xen-VMI e sulla VM sono eseguiti tutti i nuovi agenti. Il periodo che trascorre tra l'invocazione dei controlli, è stato variato da 5 a 60 secondi e, come test di paragone, è stato eseguito lo stesso benchmark disabilitando i controlli. La Fig.17 mostra che, nel caso peggiore, la degradazione delle prestazioni sul "test read" è inferiore al 6%, rispetto al caso in cui non sono effettuati i controlli. Lo stesso risultato è stato ottenuto per il "test write". Ovviamente, anche Xen di per sé introduce un overhead, se pure minimo, che dobbiamo considerare solo nel caso in cui questi risultati siano confrontati con quelli

ottenuti utilizzando la tecnica dell'introspezione a livello hardware/firmware, tramite componenti specializzate. Questo aspetto non è però compreso nel nostro lavoro.

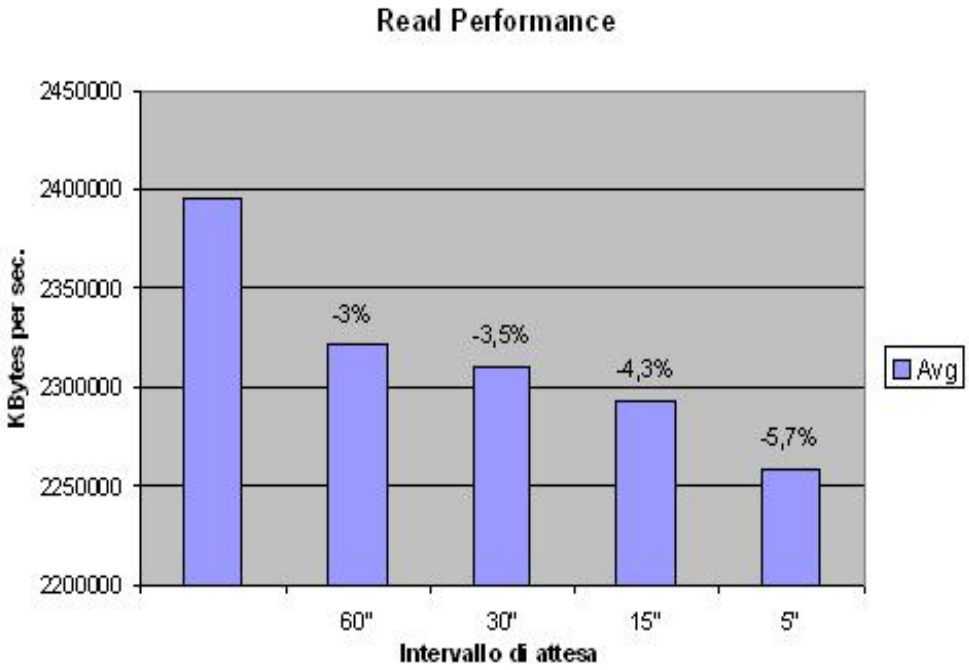


Fig.17: Test IOzone: Read Performance

Inoltre, abbiamo condotto un test per valutare quale implementazione del controllo sulla modalità promiscua sia la più efficiente. Nella sezione 3.2.1 abbiamo visto che il controllo sulla modalità promiscua richiede di esaminare un ridotto numero di informazioni, relativamente ad altri controlli, ad esempio quello sui moduli del kernel (vedi sezione 3.2.3.2). Per questo motivo, il vantaggio di un accesso diretto alle strutture dati del kernel, del modulo `antisniff`, non è sufficiente a bilanciare lo svantaggio dovuto all'esecuzione del controllo che garantisce la corretta esecuzione del modulo. In questi casi, risulta più efficiente un controllo mediante introspezione, poiché la penalità introdotta dall'utilizzo della libreria Xen-VMI è minore rispetto a quella dovuta all'utilizzo di un LKM, a cui deve essere sommato l'overhead dovuto al controllo sui moduli del kernel. Nei casi in cui si devono controllare grosse quantità di informazioni, ad esempio la text area di tutti i processi in esecuzione (vedi sezione 3.2.2), conviene ridurre al minimo le informazioni controllate mediante introspezione ed è meno penalizzante realizzare un modulo del kernel ed implementare il controllo sui moduli del kernel mediante introspezione.

La Fig.18 confronta i risultati del benchmark ottenuti eseguendo la funzione di introspezione *if_promisc()* e il modulo `antisniff`. Come si può vedere, il controllo mediante introspezione provoca un calo di performance minore rispetto a quello implementato da un modulo in cui, però, l'overhead dominante è introdotto dal controllo sui moduli del kernel. Durante i test, quest'ultimo controllo è stato eseguito ad intervalli di 30 secondi.

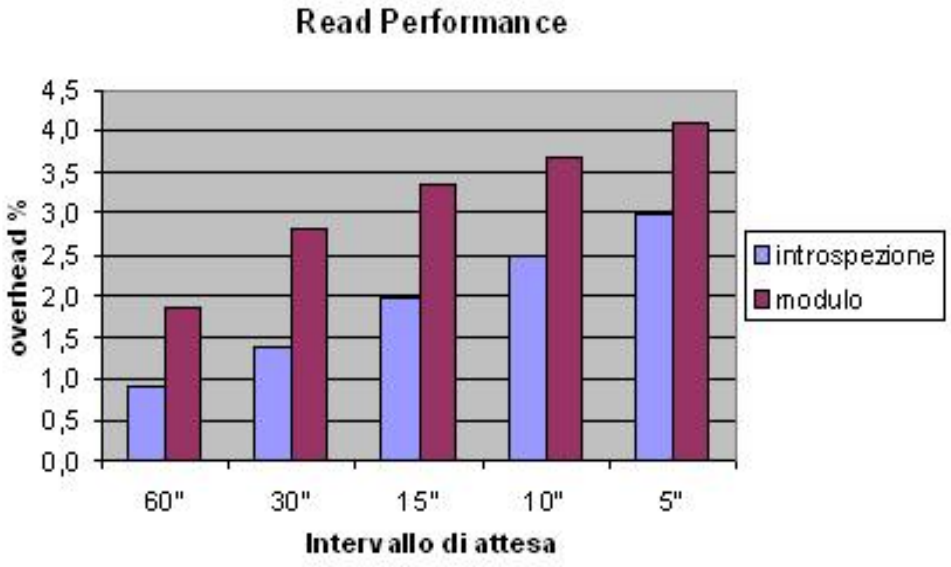


Fig.18: Confronto prestazioni: Modalità promiscua

4.3 Ottimizzazioni

È stato condotto un test per valutare l'impatto di alcune ottimizzazioni sul controllo sulla text area del kernel. La funzione di introspezione *get_kernel_text()* è più penalizzante per il dominio monitorato rispetto ad altre funzioni di introspezione, perché deve restituire al manager un alto numero di pagine del kernel, circa 450, di cui poi viene calcolato l'hash. Ricordiamo che durante la fase di introspezione viene sospesa l'esecuzione del dominio monitorato. È possibile ridurre l'overhead introdotto dal controllo, modificando la funzione *get_kernel_text()* in modo che recuperi solo un sottoinsieme casuale di tutte le pagine del kernel. Occorre però valutare anche in che misura è penalizzata l'efficacia del controllo. La Fig.19 illustra l'aumento percentuale dell'efficienza del controllo, in relazione al numero di pagine che devono essere recuperate dalla funzione di introspezione. Da notare che, l'overhead introdotto sul dominio monitorato è dovuto

unicamente alla fase di esecuzione della funzione di introspezione, la successiva fase di controllo coinvolge solo il dominio 0.

La tabella della Fig.20 mostra l'efficacia percentuale del controllo, in funzione del numero di pagine controllate rispetto al totale di 446, e del numero di pagine coinvolte in un attacco. Se si ha interesse a rilevare gli attacchi che modificano decine di pagine del kernel si può privilegiare l'efficienza, controllando un piccolo insieme di pagine, viceversa, se si vuole un alto livello di efficacia in tutte le situazioni occorre controllare almeno la metà del totale delle pagine. I valori nella tabella sono stati calcolati nel seguente modo:

$\prod_{x=(n-c+1)}^{n} \frac{x-m}{x}$	n= # pagine text area del kernel
	m= # pagine modificate da una attacco
	c= # pagine recuperate e controllate

Infine, la tabella della Fig.21 mette in relazione il livello di efficacia in un determinato contesto con il costo che richiede, cioè il numero di pagine da recuperare e controllare. Maggiore è il valore del rapporto, migliore è il compromesso.

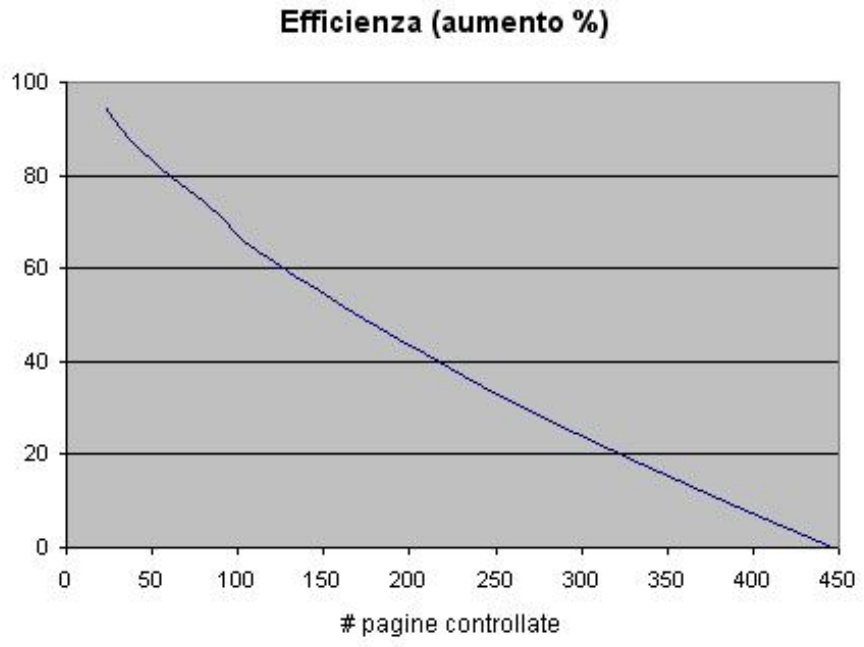


Fig.19: Ottimizzazioni - Efficienza

Efficacia %	# pagine modificate da un attacco					
# pagine controllate	20	10	5	3	2	1
446	100,0	100,0	100,0	100,0	100,0	100,0
335	100,0	100,0	100,0	98,5	93,9	75,0
223	100,0	100,0	97,0	87,6	75,0	50,0
112	99,9	94,4	76,6	58,1	44,0	25,0
89	99,0	89,5	67,3	48,2	36,0	20,0
45	88,7	65,9	41,4	27,4	19,2	10,0
22	74,5	40,0	22,4	14,1	9,7	5,0

Fig.20: Ottimizzazioni - Efficacia

Efficacia(%) / costo	1	2	3	5	10	20
446	0,22	0,22	0,22	0,22	0,22	0,22
335	0,22	0,28	0,29	0,30	0,30	0,30
223	0,22	0,34	0,39	0,43	0,45	0,45
112	0,22	0,39	0,52	0,69	0,85	0,90
89	0,22	0,40	0,54	0,75	1,00	1,11
45	0,22	0,43	0,61	0,93	1,48	1,99
22	0,22	0,43	0,63	1,00	1,79	3,34

Fig.21: Ottimizzazioni - Efficacia / Costo

5. Conclusioni

Il lavoro svolto ha riguardato la definizione e l'implementazione di un insieme di controlli per rilevare attacchi contro il kernel e le componenti a livello utente di un sistema operativo in esecuzione su una VM. Questi controlli sono realizzati mediante introspezione e sfruttano l'accesso diretto fornito dal VMM alle componenti allocate alla VM. È stata discussa la progettazione e la realizzazione di *Xen-VMI*, una libreria di funzioni per monitorare l'integrità del kernel in esecuzione su una VM da un'altra VM, mediante l'introspezione (VMI). I controlli implementati in Xen-VMI utilizzano gli *header del kernel* per colmare il "gap semantico" tra la visione del sistema dall'interno e quella ottenuta con l'introspezione. Inoltre, il controllo dei moduli del kernel garantisce anche la corretta esecuzione degli agenti *antisniff* e *check_proc*. Abbiamo dimostrato che i nuovi controlli aumentano l'efficacia del sistema Psyco-Virt, poiché rilevano molti tipi di attacchi che in precedenza non erano rilevati. Infine, è stato valutato l'overhead introdotto dai controlli di consistenza che è sempre inferiore al 6% e quindi accettabile. È possibile ridurre ulteriormente l'overhead ottimizzando le funzioni di Xen-VMI, in particolare, intervenendo sul mapping delle pagine del kernel.

Al momento, alcuni attacchi non possono essere rilevati direttamente tramite le funzioni della libreria Xen-VMI e gli altri controlli. Ad esempio, oltre alla tabella dei puntatori alle chiamate di sistema o alla tabella delle interruzioni, vi sono altre regioni critiche nel kernel che devono essere protette, ad esempio le strutture dati del Virtual File System [40], gli handler per la gestione dei Page Fault [3], ed altre ancora [2]. Anche una qualsiasi modifica illecita ai dati dinamici in memoria, come ad esempio allo stack del kernel, non è rilevata. La complessità di prevenire e rilevare questo tipo di attacchi è molto alta, perché è necessario calcolare a priori un insieme di invarianti relative a queste strutture dati, in modo che il manager, a tempo di esecuzione, possa verificare la loro validità. Un altro problema nasce se l'attaccante è a conoscenza del fatto che il sistema operativo è in esecuzione all'interno di una macchina virtuale. Infatti, l'attaccante può tentare o di attaccare direttamente il VMM [7,21] oppure di eludere i controlli periodici di consistenza [33]. Nel caso dell'evasione dei controlli, un attaccante può modificare le strutture dati critiche del kernel non appena i controlli di consistenza sono stati eseguiti e, successivamente, ripristinare il sistema ad uno stato consistente prima che questi controlli siano nuovamente effettuati, evitando così di essere rilevato. Infine, se un attaccante riuscisse a modificare il kernel associato ad una VM prima che essa sia messa in

esecuzione, alcuni attacchi non sarebbero rilevati. Ad esempio, un modulo con finalità illecite potrebbe essere considerato autentico e quindi autorizzato ad essere eseguito.

Riferimenti

[1] Arati Baliga, Xiaoxin Chen, and Liviu Iftode. Paladin: Automated detection and containment of rootkit attacks, Jan 2006. Rutgers University Department of Computer Science Technical Report DCSTR593.

[2] Arati Baliga, Pandurang Kamat, and Liviu Iftode, Lurking in the shadows: Identifying systemic threats to kernel data, SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy (Washington, DC, USA), IEEE Computer Society, 2007.

[3] buffer, Hijacking Linux Page Fault Handler, Phrack 11(61), 2003.

[4] chkrootkit – locally checks for signs of a rootkit. http://www.chkrootkit.org/.

[5] Debian – The Universal Operating System. http://www.debian.org/.

[6] Fedora project. http://fedora.redhat.com/.

[7] P. Ferrie, Attacks on Virtual Machine Emulators, Symantec Security Response 5 (2006).

[8] T Fraser. Automatic discovery of integrity constraints in binary kernel modules, Tech. report, University of Maryland Institute for Advanced Computer Studies, December 2004.

[9] The FU rootkit. http://www.rootkit.com/project.php?id=12.

[10] FuSyS. Kstat. http://www.s0ftpj.org/tools/kstat24_v1.1-2.tgz.

[11] Julian B. Grizzard, John G. Levine, and Henry L. Owen, Detecting and Categorizing Kernel-Level Rootkit to Aid Future Detection. Host Security, 2006.

[12] grsecurity, http://www.grsecurity.net/.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Proc. Network and Distributed Systems Security Symposium, February 2003.

[14] Vincenzo Giacchina. Sniffing e programmazione C a basso livello. Pluto Ware, 2001.

[15] halflife, Abuse of the Linux Kernel for Fun and Profit, Phrack, 7(50), 1997.

[16] Gianluca Insolvibile. The Linux Socket Filter: Sniffing Bytes over the Network. Copyright Linux Journal, 2001.

[17] Intel Virtualization Technology. http://www.intel.com/technology/computing/vptech/.

[18] IOzone Filesystem Benchmark. http://www.iozone.org/.

[19] X. Jiang, X. Wang, and D. Xu, Stealthy malware detection through vmm-based 'out-of the-box' semantic view reconstruction, Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007), November 2007.

[20] Ashlesha Joshi, Samuel T King, George W Dunlap, and Peter M Chen. Detecting past and present intrusions through vulnerability specific predicates. In SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, New York, NY, USA, 2005. ACM Press.

[21] Sebastien Josse. Rootkit detection from outside the Matrix. EICAR 2007 Best Academic Paper, France, 2007.

[22] Bryan Henderson. Linux Loadable Kernel Module HOWTO. Linux Documentation Project, 2001, Revision v.1.09, 2006.

[23] kad. Handling Interrupt Descriptor Table for fun and profit. Phrack, 11(59), July 2002.

[24] John Levine, Julian Grizzard, and Henry Owen, A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table, IWIA '04: Proceedings of the Second IEEE International Information Assurance Workshop(IWIA'04) (Washington, DC, USA), IEEE Computer Society, 2004.

[25] LIDS project: LIDS Secure Linux System. http://www.lids.org/.

[26] David Larochelle, David Evans. Staticaly Detecting Likely Buffer Overflow Vulnerabilities. University of Virginia, Department of Computer Science, 2001.

[27] Robert Love, Linux Kernel Development 2nd Edition. Sams Publishing, 2005.

[28] G. Mazzei, A. Paolessi, S. Volpini. Buffer Overflow. Corso di Sistemi Operativi, Facoltà di Ingegneria, Università degli Studi di Siena, 2002.

[29] Openssl: The open source toolkit for ssl/tls. http://www.openssl.org/.

[30] OpenVPN - An Open Source SSL VPN Solution. http://openvpn.net/.

[31] Open source tripwire. http://sourceforge.net/projects/tripwire/.

[32] Nick L. Petroni, Timothy Fraser, Jesus Molina and William A. Arbaugh. Copilot - a Coprocessor-based kernel runtime integrity monitor. In USENIX Security Symposium, 2004.

[33] Nick L. Petroni, Jr. and Michael Hicks. Automated Detections of Persistent Kernel Control-Flow Attacks. CCS'07, 2007.

[34] The Python programming language. http://www.python.org/.

[35] Nguyen Anh Quynh and Yoshiyasu Takefuji, Towards a tamper resistant kernel rootkit detector, SAC '07: Proceedings of the 2007 ACM symposium on Applied computing (New York, NY, USA), ACM Press, 2007.

[36] Peter J. Salzman, Micheal Burian, Ori Pomerantz. The Linux Kernel Module Programming Guide. Copyright 2001 Peter J. Salzman, Version 2.6.4, 2007.

[37] Daniele Scandurra, Architetture di sicurezza e tecnologie di virtualizzazione: Rilevamento delle intrusioni tramite introspezione. Università di Pisa, Italy, 2006.

[38] sd and devik. Linux on-the-fly kernel patching without LKM. Phrack, 10(58), 2001.

[39] Security-Enhanced Linux. http://www.nsa.gov/selinux/.

[40] Alkesh Shah, Analysis of rootkits: Attack approaches and detection mechanisms, Tech. Report, Georgia Institute of Technology.

[41] Stefan Klaas, Sovrascrivere una funzione con `ptrace()`. Hackin9, No. 3, 2007.

[42] Tcpdump. Network Sniffer/Analyzer for UNIX. http://www.tcpdump.org/.

[43] Claudio Telmon, Fabrizio Baiardi. I Sistemi di Intrusion Detection, 2005.

[44] University of Cambridge UK. Xen interface manual: Xen v3.0 for x86.

[45] University of Cambridge UK. The Xen virtual machine monitor.

[46] VMware. http://www.vmware.com/.

[47] Lifu Wang and Partha Dasgupta. Kernel and application integrity assurance: Ensuring freedom from rootkits and malware in a computer system. In AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops,Washington, DC, USA, 2007. IEEE Computer Society.

[48] XenAccess Library. http://xenaccess.sourceforge.net/.

[49] Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang, Towards a vmm-based usage control framework for OS kernel integrity protection, SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies (New York, NY, USA), ACM Press, 2007.

Appendice

In questa sezione viene riportato il codice, opportunamente commentato, di tutti i controlli implementati. La descrizione è divisa in tre sezioni: nella prima viene descritto il codice delle funzioni di introspezione e del programma che effettua i controlli di consistenza, nella seconda è descritto il codice dei moduli del kernel e nella terza è descritto il codice del manager e degli agenti "wrapper".

A. Xen-VMI

```
/*
 * xen_vmi_private.h
 *
 * Dichiarazioni utili per le funzioni di introspezione
 */
#include <stdio.h>
#include <linux/string.h>
#include <linux/list.h>
#include <linux/types.h>
#include <linux/module.h>            /* per i moduli del kernel */
#include <linux/sched.h>             /* per i processi */
#include <linux/netdevice.h>         /* per le interfacce di rete */
#include <linux/file.h>              /* per i file */
#include <linux/dcache.h>
#include <linux/fs.h>
#include <linux/stat.h>
#include <linux/spinlock.h>
#include <linux/spinlock_types.h>

#define MODULE_NAME_LEN (64 - sizeof(unsigned long))
#define SHA1_DIGEST_SIZE 20
#define IDT_ENTRIES 256
/*dichiarazioni per le funzioni di XenAccess */
#define XC_PAGE_SHIFT 12
#define XC_PAGE_SIZE (1UL << XC_PAGE_SHIFT)
#define XA_SUCCESS 0
#define XA_FAILURE -1
#define XA_OS_LINUX 0
#define XA_OS_WINDOWS 1  /* not yet supported */
#define XA_OS_NETBSD 2   /* not yet supported */
#define XA_PAGE_SIZE XC_PAGE_SIZE
#define XA_PAGE_OFFSET 0xc0000000

#define next_module(m)list_entry((m)->list.next, struct module, list)
#define prev_module(m)list_entry((m)->list.prev, struct module, list)


typedef uint8_t xen_domain_handle_t[16];
typedef struct xc_dominfo
{
     uint32_t      domid;
     uint32_t      ssidref;
     unsigned int  dying:1, crashed:1, shutdown:1,
         paused:1, blocked:1, running:1,
         hvm:1;
     unsigned int  shutdown_reason; /* only meaningful if shutdown==1 */
     unsigned long nr_pages;
     unsigned long shared_info_frame;
     uint64_t      cpu_time;
     unsigned long max_memkb;
     unsigned int  nr_online_vcpus;
```

```
     unsigned int  max_vcpu_id;
     xen_domain_handle_t handle;
} xc_dominfo_t;

/*per memorizzare le informazioni di controllo sui moduli */
struct mod_info
{
     char mod_name[MODULE_NAME_LEN];
     unsigned char *mod_text;
     unsigned long mod_size;
     int license;
};

/*per memorizzare i digest */
struct digest_info
{
     char name[MODULE_NAME_LEN];
     unsigned char digest[SHA1_DIGEST_SIZE];
};

/*trap_table di Xen */
typedef struct trap_info
{
     uint8_t       vector;  /* exception vector                              */
     uint8_t       flags;   /* 0-3: privilege level; 4: clear event enable?  */
     uint16_t      cs;      /* code selector                                 */
     unsigned long address; /* code offset                                   */
} trap_info_t;

/*per interfacciarsi con la libreria di controllo di Xen */
typedef struct xa_instance
{
     int xc_handle;          /* handle to xenctrl library (libxc) */
     uint32_t domain_id;     /* domid that we are accessing */
     char *domain_name;      /* domain name that we are accessing */
     char *sysmap;           /* system map file for domain's running kernel */
     uint32_t kpgd;          /* kernel page global directory */
     uint32_t init_task;     /* address of task struct for init */
     int os_type;            /* type of os: XA_OS_LINUX, etc */
     int hvm;                /* nonzero if HVM domain */
     xc_dominfo_t info;      /* libxc info: domid, ssidref, stats, etc */
     unsigned long *live_pfn_to_mfn_table;
     unsigned long nr_pfns;
} xa_instance_t;

/*funzioni di interfaccia di XenAccess */
int xa_init (uint32_t domain_id, xa_instance_t *instance);
void *xa_access_kernel_symbol (
               xa_instance_t *instance, char *symbol, uint32_t *offset);
int xa_destroy (xa_instance_t *instance);
int linux_system_map_symbol_to_address (
               xa_instance_t *instance, char *symbol, uint32_t *address);
void *xa_access_virtual_address (
               xa_instance_t *instance, uint32_t virt_address, uint32_t *offset);
/*funzioni della libreria di controllo di Xen */
extern int xc_domain_pause(int xc_handle, uint32_t domid);
extern int xc_domain_unpause(int xc_handle,  uint32_t domid);
/*funzioni C standard */
extern int atoi (__const char *__nptr);
extern int munmap (void *__addr, size_t __len);
extern void *malloc(size_t size);
extern void *realloc(void *ptr, size_t size);
extern void free(void *ptr);
extern unsigned int sleep(unsigned int seconds);

/*per ricostruire il path di un file */
#define MAX_LENGTH 1000
struct dentry_list
{
     char name[MAX_LENGTH];
```

```
     struct dentry_list *previous;
};
struct dentry_list *last_dentry = NULL;



/*
 * xen_vmi.c
 *
 * Funzioni di introspezione
 */
#include "xen_vmi_private.h"


/*ricostruisce la lista dei moduli del kernel e restituisce il puntatore ad una
  lista contenente tutte le informazioni utili da controllare */
int get_modules_infos(struct mod_info ***info_ptr, int *num_modules, uint32_t dom)
{
     xa_instance_t xai;
     unsigned char *memory = NULL;
     unsigned char *memoryinfo = NULL;
     uint32_t offset, next_addr, end_addr, tmp_offset;
     struct module *mod;
     char *symbol = "modules";
     struct list_head *head_ptr;
     /*virtul addresses of list_head structs*/
     uint32_t first_next, next_next;
     struct mod_info *elem = NULL;
     int i = 0;


     if(xa_init(dom, &xai) == XA_FAILURE)
     {
         perror("failed to init XenAccess library");
         goto error_exit;
     }

     memory = xa_access_kernel_symbol(&xai, symbol, &offset);
     if(NULL == memory)
     {
         perror("failed to get modules list head");
         goto error_exit;
     }

     if(linux_system_map_symbol_to_address(
                             &xai, symbol, &first_next) == XA_FAILURE)
     {
         perror("failed to get address of first module");
         goto error_exit;
     }

     head_ptr = (struct list_head *)(memory + offset);
     munmap(memory, XA_PAGE_SIZE);

     xc_domain_pause(xai.xc_handle, dom); /* esecuzione del dominio sospesa */

     memory = xa_access_virtual_address(&xai, first_next, &offset);
     head_ptr = (struct list_head *)(memory + offset);

     mod = list_entry(head_ptr, struct module, list);
     next_addr = (uint32_t)next_module(mod);
     next_next = (uint32_t)mod->list.next;

     /*scorrimento della lista bidirezionale e circolare */
     while(first_next != next_next)
     {
         munmap(memory, XA_PAGE_SIZE);
         memory = xa_access_virtual_address(&xai, next_addr, &offset);
         mod = (struct module*)(memory + offset);
```

```
/*stampa a video delle informazioni sui moduli: nome, size e licenza */
if(strlen(mod->name) < 6) printf("[%s]\t\t",mod->name);
else printf("[%s]\t",mod->name);
printf("%lu\t", (mod->init_size + mod->core_size));
if(!mod->license_gplok) printf("NOT-GPL\n");
else printf("GPL\n");

/*recupero della core_text_section */
next_addr = (uint32_t)mod->module_core;
memoryinfo = xa_access_virtual_address(&xai, next_addr, &offset);
elem = (struct mod_info *)malloc(sizeof(struct mod_info));
strncpy(elem->mod_name,mod->name,MODULE_NAME_LEN);
elem->license = mod->license_gplok;
elem->mod_size = mod->init_text_size + mod->core_text_size;
elem->mod_text =
     (unsigned char *)malloc(elem->mod_size * sizeof(unsigned char));
tmp_offset = (uint32_t)(XA_PAGE_SIZE - offset);
if((uint32_t)mod->core_text_size > tmp_offset)
{
     end_addr = (uint32_t)(next_addr + mod->core_text_size);
     memcpy(elem->mod_text, memoryinfo+offset, (size_t)tmp_offset);
     next_addr = next_addr + tmp_offset;
     while((end_addr-next_addr) > (uint32_t)XA_PAGE_SIZE)
     {
          munmap(memoryinfo, XA_PAGE_SIZE);
          memoryinfo = xa_access_virtual_address(
                                        &xai, next_addr, &offset);
          memcpy((elem->mod_text + tmp_offset),
                                    memoryinfo+offset, XA_PAGE_SIZE);
          tmp_offset = tmp_offset + (uint32_t)XA_PAGE_SIZE;
          next_addr = (uint32_t)(next_addr + XA_PAGE_SIZE);
     }
     munmap(memoryinfo, XA_PAGE_SIZE);
     memoryinfo = xa_access_virtual_address(&xai, next_addr, &offset);
     memcpy((elem->mod_text + tmp_offset),
                    memoryinfo+offset, (size_t)(end_addr-next_addr));
}
else memcpy(elem->mod_text, memoryinfo+offset, mod->core_text_size);

/*recupero della init_text_section, se non nulla */
if(mod->init_text_size > 0)
{
     if(memoryinfo) munmap(memoryinfo, XA_PAGE_SIZE);
     next_addr = (uint32_t)mod->module_init;
     memoryinfo = xa_access_virtual_address(&xai, next_addr, &offset);
     tmp_offset = (uint32_t)(XA_PAGE_SIZE - offset);
     if((uint32_t)elem->mod_size > tmp_offset)
     {
          end_addr = (uint32_t)(next_addr + mod->init_text_size);
          memcpy((elem->mod_text + mod->core_text_size),
                              memoryinfo+offset, (size_t)tmp_offset);
          next_addr = next_addr + tmp_offset;
          while((end_addr-next_addr) > (uint32_t)XA_PAGE_SIZE)
          {
               munmap(memoryinfo, XA_PAGE_SIZE);
               memoryinfo = xa_access_virtual_address(
                                        &xai, next_addr, &offset);
               memcpy((elem->mod_text + mod->core_text_size + tmp_offset),
                                    memoryinfo+offset, XA_PAGE_SIZE);
               tmp_offset = tmp_offset + (uint32_t)XA_PAGE_SIZE;
               next_addr = (uint32_t)(next_addr + XA_PAGE_SIZE);
          }
          munmap(memoryinfo, XA_PAGE_SIZE);
          memoryinfo = xa_access_virtual_address(
                                        &xai, next_addr, &offset);
          memcpy((elem->mod_text + mod->core_text_size + tmp_offset),
                    memoryinfo+offset, (size_t)(end_addr-next_addr));
     }
     else memcpy((elem->mod_text + mod->core_text_size),
                              memoryinfo+offset, mod->init_text_size);
```

```
        }
       /*inserimento delle informazioni su ogni modulo nella lista da ritornare*/
        if(!i)
             (*info_ptr) = (struct mod_info **)malloc(sizeof(struct mod_info *));
        else (*info_ptr) = (struct mod_info **)realloc(
                           (*info_ptr), (i+1) * sizeof(struct mod_info *));
        (*info_ptr)[i] = elem;
        i++;
        if(memoryinfo) munmap(memoryinfo, XA_PAGE_SIZE);

    }
    *num_modules = i; /*numero di moduli registrati */
    xc_domain_unpause(xai.xc_handle, dom); /*esecuzione del dominio riavviata */

error_exit:
    xa_destroy(&xai);
    if(memory) munmap(memory, XA_PAGE_SIZE);

    return 0;
}


/*restituisce il puntatore al contenuto della text area del kernel */
int get_kernel_text(unsigned char ***text_page, int *page_num, uint32_t dom)
{
    xa_instance_t xai;
    unsigned char *memory = NULL;
    uint32_t offset;
    char *symbol_start = "_text";
    char *symbol_end = "_etext";
    uint32_t start_addr, end_addr, next_addr;


    (*page_num)=0; /*numero di pagine del kernel da mappare */
    if(xa_init(dom, &xai) == XA_FAILURE)
    {
        perror("failed to init XenAccess library");
        goto error_exit;
    }

    if(linux_system_map_symbol_to_address(
                           &xai, symbol_start, &start_addr) == XA_FAILURE)
    {
        perror("failed to get start addr");
        goto error_exit;
    }

    if(linux_system_map_symbol_to_address(
                                  &xai, symbol_end, &end_addr) == XA_FAILURE)
    {
        perror("failed to get end addr");
        goto error_exit;
    }

    next_addr = start_addr;
    memory = xa_access_virtual_address(&xai, next_addr, &offset);
    (*page_num)++;
    (*text_page)=(unsigned char **)(malloc(*page_num * sizeof(unsigned char *)));
    (*text_page)[*page_num-1] =
             (unsigned char *)(malloc(XA_PAGE_SIZE * sizeof(unsigned char )));
    memcpy((*text_page)[*page_num-1], memory, XA_PAGE_SIZE);

    /*ciclo che recupera tutte le pagine del kernel relative alla text area */
    while((next_addr = next_addr + (XA_PAGE_SIZE - offset)) < end_addr)
    {
        (*page_num)++;
        munmap(memory, XA_PAGE_SIZE);

        memory = xa_access_virtual_address(&xai, next_addr, &offset);
        (*text_page)=realloc(*text_page, *page_num * sizeof(unsigned char *) );
```

```
            (*text_page)[*page_num-1] =
                 (unsigned char *)(malloc(XA_PAGE_SIZE * sizeof(unsigned char )));
            /*copia della pagina i-esima */
            memcpy((*text_page)[*page_num-1], memory, XA_PAGE_SIZE);
        }

error_exit:
        xa_destroy(&xai);
        if(memory) munmap(memory, XA_PAGE_SIZE);

        return 0;
}


/*restituisce il puntatore al contenuto della System Call Table */
int get_syscall_table(uint32_t **sys_table, int *sys_num, uint32_t dom)
{
        xa_instance_t xai;
        unsigned char *memory = NULL;
        uint32_t offset;
        char *symbol = "sys_call_table";
        uint32_t *sys_call_table;
        int i=0;


        if(xa_init(dom, &xai) == XA_FAILURE)
        {
            perror("failed to init XenAccess library");
            goto error_exit;
        }

        memory = xa_access_kernel_symbol(&xai, symbol, &offset);
        if(NULL == memory)
        {
            perror("failed to get process list head");
            goto error_exit;
        }

        sys_call_table = (uint32_t *)(memory+offset);
        *sys_table = (uint32_t *)(malloc((NR_syscalls) * sizeof(uint32_t)));
        *sys_num = NR_syscalls-1;

        for(i=0; i< NR_syscalls; i++)
        {
            /*ogni posizione della tabella contiene un indirizzo che viene copiato
              nella lista da restituire */
            (*sys_table)[i] = (uint32_t)sys_call_table[i];
        }
        munmap(memory, XA_PAGE_SIZE);

error_exit:
        xa_destroy(&xai);
        if(memory) munmap(memory, XA_PAGE_SIZE);

        return 0;
}


/*restituisce il puntatore al contenuto della trap_table, che è la tabella delle
  interruzioni gestita da Xen */
int get_idt_table(unsigned char ***idt_table, int *idt_entries, uint32_t dom)
{
        xa_instance_t xai;
        unsigned char *memory = NULL;
        uint32_t offset;
        char *symbol = "trap_table";
        trap_info_t *trap_table;
        int i = 0;
```

```
    if(xa_init(dom, &xai) == XA_FAILURE)
    {
        perror("failed to init XenAccess library");
        goto error_exit;
    }

    memory = xa_access_kernel_symbol(&xai, symbol, &offset);
    if(NULL == memory)
    {
        perror("failed to get process list head");
        goto error_exit;
    }

    trap_table = (trap_info_t *)(memory+offset);
    (*idt_table) = (unsigned char **)malloc(sizeof (unsigned char *));
    (*idt_table)[i] = (unsigned char *)malloc(sizeof (trap_info_t ));
    memcpy((*idt_table)[i], trap_table, (sizeof (trap_info_t )));
    trap_table++;

    /*ciclo che recupera tutti gli indirizzi presenti nella trap_table */
    while(++i < IDT_ENTRIES)
    {
        (*idt_table) =
        (unsigned char **)realloc((*idt_table), (i+1) * sizeof(unsigned char *));
        (*idt_table)[i] = (unsigned char *)malloc(sizeof (trap_info_t ));
        /*copia dell'indirizzo i-esimo */
        memcpy((*idt_table)[i], trap_table, (sizeof (trap_info_t )));
        trap_table++;
    }
    munmap(memory, XA_PAGE_SIZE);

error_exit:
    xa_destroy(&xai);
    if (memory) munmap(memory, XA_PAGE_SIZE);

    return 0;
}


/*controlla se le interfaccie di rete sono in modalità promiscua e segnala la
  cosa stampando un messaggio a video */
int if_promisc(uint32_t dom)
{
    xa_instance_t xai;
    unsigned char *memory = NULL;
    uint32_t offset, next_addr;
    struct net_device *dev;
    struct net_device **dev_p;


    if(xa_init(dom, &xai) == XA_FAILURE)
    {
        perror("failed to init XenAccess library");
        goto error_exit;
    }

    memory = xa_access_kernel_symbol(&xai, "dev_base", &offset);
    if(NULL == memory)
    {
        perror("failed to get process list head");
        goto error_exit;
    }

    dev_p = (struct net_device **)(memory + offset);
    next_addr = (uint32_t)(*dev_p);

    /*ciclo che scorre la lista dei dispositivi di rete */
    do {
        munmap(memory, XA_PAGE_SIZE);
        memory = xa_access_virtual_address(&xai, next_addr, &offset);
```

```
          dev = (struct net_device*)(memory + offset);
          if(strstr(dev->name,"eth") != NULL)
          {
               if((dev->flags & IFF_PROMISC) ||
                       (dev->gflags && IFF_PROMISC) || dev->promiscuity)
                   printf("%s: PROMISC\n", dev->name);
               else printf("%s: not promisc\n", dev->name);
          }

     } while((next_addr=(uint32_t)dev->next) != 0);

error_exit:
     xa_destroy(&xai);
     if(memory) munmap(memory, XA_PAGE_SIZE);

     return 0;
}


/*ricostruisce la lista dei processi attivi e la restituisce a video */
int get_process_list(uint32_t dom)
{
     xa_instance_t xai;
     unsigned char *memory = NULL;
     uint32_t offset;
     struct task_struct *task;
     /*virtual address of the first and next task_stuct */
     uint32_t init_addr, next_addr;
     char *symbol = "init_task";


     if(xa_init(dom, &xai) == XA_FAILURE) goto error_exit;

     memory = xa_access_kernel_symbol(&xai, symbol, &offset);
     if(NULL == memory) goto error_exit;

     if(linux_system_map_symbol_to_address(
            &xai, symbol, &init_addr) == XA_FAILURE) goto error_exit;

     xc_domain_pause(xai.xc_handle, dom); /* esecuzione del dominio sospesa */

     /*ciclo che scorre la “task list” */
     while(next_addr != init_addr)
     {
          task = (struct task_struct *)(memory + offset);
          next_addr = (uint32_t)next_task(task);
          /*queste informazioni sono utilizzate dal manager */
          if(task->pid) printf("%d\t%d\t%s\n", task->pid, task->uid, task->comm);
          munmap(memory, XA_PAGE_SIZE);
          memory = xa_access_virtual_address(&xai, next_addr, &offset);
     }
     xc_domain_unpause(xai.xc_handle, dom); /* esecuzione del dominio riavviata */

error_exit:
     xa_destroy(&xai);
     if(memory) munmap(memory, XA_PAGE_SIZE);

     return 0;
}


/*per formattare una parte delll’output della funzione list_open_files() */
void get_ftype(mode_t mode, char *type)
{
     if(S_ISREG (mode))
          snprintf(type, 4, "REG");
     else if(S_ISDIR (mode))
          snprintf(type, 4, "DIR");
     else if(S_ISLNK (mode))
          snprintf(type, 5, "LINK");
```

```
     else if(S_ISFIFO (mode))
          snprintf(type, 5, "FIFO");
     else if(S_ISSOCK (mode))
          snprintf(type, 5, "SOCK");
     else if(S_ISBLK (mode))
          snprintf(type, 6, "BLOCK");
     else if(S_ISCHR (mode))
          snprintf(type, 5, "CHAR");
     else
          snprintf(type, 8, "unknown");
}


/*ricostruisce la lista dei file aperti e la restituisce a video */
int list_open_files(uint32_t dom)
{
     xa_instance_t xai;
     unsigned char *memory = NULL;
     uint32_t offset;
     struct task_struct *task;
     /*virtual address of the first and next task_stuct */
     uint32_t init_addr, next_addr;
     char *symbol = "init_task";


     if(xa_init(dom, &xai) == XA_FAILURE) goto error_exit;

     memory = xa_access_kernel_symbol(&xai, symbol, &offset);
     if(NULL == memory) goto error_exit;

     if (linux_system_map_symbol_to_address(
               &xai, symbol, &init_addr) == XA_FAILURE) goto error_exit;

     xc_domain_pause(xai.xc_handle, dom); /* esecuzione del dominio sospesa */

     /*ciclo che scorre la “task list” */
     while(next_addr != init_addr)
     {
          uint32_t files_addr;
          uint32_t fdt_addr;
          uint32_t fd_addr;
          unsigned char *memory_file= NULL;
          unsigned char *memory_fdt = NULL;
          unsigned char *memory_fd = NULL;
          struct files_struct *files;
          struct fdtable *fdt;
          struct files **fd;

          int pid;
          char type[10];
          unsigned long node;
          char file_name[MAX_LENGTH];

          task = (struct task_struct *)(memory + offset);
          next_addr = (uint32_t)next_task(task);
          pid = task->pid;

          /*file mappati */
          if(task->mm != NULL)
          {
               uint32_t mm_addr;
               unsigned char *memory_mm;
               struct mm_struct *mm;
               uint32_t vm_addr;
               unsigned char *memory_vm;
               struct vm_area_struct *vm;

               mm_addr = (uint32_t)task->mm;
               memory_mm = xa_access_virtual_address(&xai, mm_addr, &offset);
               mm = (struct mm_struct *)(memory_mm + offset);
```

```
vm_addr = (uint32_t)mm->mmap;
munmap(memory_mm,XA_PAGE_SIZE);
while(vm_addr != 0)
{
     uint32_t next_file_addr;
     unsigned char *memory_next_file;
     struct file *next_file;
     uint32_t next_dentry_addr;
     unsigned char *memory_dentry;
     struct dentry *f_dentry;
     uint32_t next_filename_addr;
     unsigned char *memory_next_filename;
     char *next_filename;
     uint32_t last_dentry_addr;
     uint32_t next_inode_addr;
     unsigned char *memory_inode;
     struct inode *next_inode;

     memory_vm = xa_access_virtual_address(&xai, vm_addr, &offset);
     vm = (struct vm_area_struct *)(memory_vm + offset);

     next_file_addr = (uint32_t)(vm->vm_file);
     if(vm->vm_file != NULL)
     {
          memory_next_file = xa_access_virtual_address(
                              &xai, next_file_addr, &offset);
          next_file = (struct file *)(memory_next_file + offset);
          next_dentry_addr = (uint32_t)(next_file->f_dentry);
          memory_dentry = xa_access_virtual_address(
                              &xai, next_dentry_addr, &offset);
          f_dentry = (struct dentry *)(memory_dentry + offset);
          next_filename_addr = (uint32_t)(f_dentry->d_name.name);
          memory_next_filename = xa_access_virtual_address(
                              &xai, next_filename_addr, &offset);
          next_filename = (char *)(memory_next_filename + offset);
          next_inode_addr = (uint32_t)(f_dentry->d_inode);
          memory_inode = xa_access_virtual_address(
                              &xai, next_inode_addr, &offset);
          next_inode = (struct inode *)(memory_inode + offset);
          node = next_inode->i_ino;
          get_ftype(next_inode->i_mode, type);
          munmap(memory_inode, XA_PAGE_SIZE);
          last_dentry =
          (struct dentry_list *)(malloc(sizeof(struct dentry_list)));
          last_dentry->previous = NULL;
          strncpy(last_dentry->name, next_filename, MAX_LENGTH-1);
          last_dentry_addr = next_dentry_addr;
          next_dentry_addr = (uint32_t)(f_dentry->d_parent);

          while(last_dentry_addr != next_dentry_addr)
          {
               struct dentry_list *new_dentry = NULL;
               munmap(memory_dentry, XA_PAGE_SIZE);
               memory_dentry = xa_access_virtual_address(
                              &xai, next_dentry_addr, &offset);
               f_dentry = (struct dentry *)(memory_dentry + offset);
               last_dentry_addr = next_dentry_addr;
               next_dentry_addr = (uint32_t)(f_dentry->d_parent);
               next_filename_addr =
                              (uint32_t)(f_dentry->d_name.name);
               munmap(memory_next_filename, XA_PAGE_SIZE);
               memory_next_filename = xa_access_virtual_address(
                              &xai, next_filename_addr, &offset);
               next_filename =
                        (char *)(memory_next_filename + offset);
               new_dentry =
             (struct dentry_list *)(malloc(sizeof(struct dentry_list )));
               new_dentry->previous = last_dentry;
               last_dentry = new_dentry;
               strncpy(new_dentry->name,next_filename,MAX_LENGTH-1);
```

```
                    }
                    file_name[0] = '\0';
                    if(!(strstr(type, "SOCK")))
                    {
                         while(last_dentry != NULL)
                         {
                              struct dentry_list *old_dentry = last_dentry;
                              if(strncmp(last_dentry->name, "/", MAX_LENGTH-1))
                              {
                                   strcat(file_name, last_dentry->name);
                                   strcat(file_name, "/");
                              }
                              else strcat(file_name, "/");
                              last_dentry = last_dentry->previous;
                              free(old_dentry);
                         }
                         file_name[strlen(file_name) -1] = '\0';
                    }
                    else strcat(file_name, "/");
                    free(last_dentry);
                    if(!strcmp(type,"REG"))
                    {
                          /*queste informazioni sono utilizzate dal manager */
                          printf("%d#%lu#%s\n", pid, node, file_name);
                    }
                    munmap(memory_next_file, XA_PAGE_SIZE);
                    munmap(memory_dentry, XA_PAGE_SIZE);
                    munmap(memory_next_filename, XA_PAGE_SIZE);
               }
               vm_addr = (uint32_t)vm->vm_next;
               munmap(memory_vm,XA_PAGE_SIZE);

          }
     }
     files_addr = (uint32_t)task->files;
     memory_file = xa_access_virtual_address(&xai, files_addr, &offset);
     files = (struct files_struct *)(memory_file + offset);

     fdt_addr = (uint32_t)files->fdt;
     memory_fdt= xa_access_virtual_address(&xai, fdt_addr, &offset);
     fdt = (struct fdtable *)(memory_fdt + offset);

     fd_addr = (uint32_t)fdt->fd;
     memory_fd = xa_access_virtual_address(&xai, fd_addr, &offset);
     fd = (struct files **)(memory_fd + offset);

     /*file aperti */
     while(*fd)
     {
          uint32_t next_file_addr;
          unsigned char *memory_next_file;
          struct file *next_file;
          uint32_t next_dentry_addr;
          unsigned char *memory_dentry;
          struct dentry *f_dentry;
          uint32_t next_filename_addr;
          unsigned char *memory_next_filename;
          char *next_filename;
          uint32_t last_dentry_addr;
          uint32_t next_inode_addr;
          unsigned char *memory_inode;
          struct inode *next_inode;

          next_file_addr = (uint32_t)(*fd);
          memory_next_file = xa_access_virtual_address(
                                        &xai, next_file_addr, &offset);
          next_file = (struct file *)(memory_next_file + offset);
          next_dentry_addr = (uint32_t)(next_file->f_dentry);
          memory_dentry = xa_access_virtual_address(
                                        &xai, next_dentry_addr, &offset);
```

```
        f_dentry = (struct dentry *)(memory_dentry + offset);
        next_filename_addr = (uint32_t)(f_dentry->d_name.name);
        memory_next_filename = xa_access_virtual_address(
                                    &xai, next_filename_addr, &offset);
        next_filename = (char *)(memory_next_filename + offset);
        next_inode_addr = (uint32_t)(f_dentry->d_inode);
        memory_inode = xa_access_virtual_address(
                                    &xai, next_inode_addr, &offset);
        next_inode = (struct inode *)(memory_inode + offset);
        node = next_inode->i_ino;
        get_ftype(next_inode->i_mode, type);
        munmap(memory_inode, XA_PAGE_SIZE);
        last_dentry =
               (struct dentry_list *)(malloc(sizeof(struct dentry_list )));
        last_dentry->previous = NULL;
        strncpy(last_dentry->name, next_filename, MAX_LENGTH-1);
        last_dentry_addr = next_dentry_addr;
        next_dentry_addr = (uint32_t)(f_dentry->d_parent);

        while(last_dentry_addr != next_dentry_addr)
        {
             struct dentry_list *new_dentry = NULL;
             munmap(memory_dentry, XA_PAGE_SIZE);
             memory_dentry = xa_access_virtual_address(
                                    &xai, next_dentry_addr, &offset);
             f_dentry = (struct dentry *)(memory_dentry + offset);
             last_dentry_addr = next_dentry_addr;
             next_dentry_addr = (uint32_t)(f_dentry->d_parent);
             next_filename_addr = (uint32_t)(f_dentry->d_name.name);
             munmap(memory_next_filename, XA_PAGE_SIZE);
             memory_next_filename = xa_access_virtual_address(
                                    &xai, next_filename_addr, &offset);
             next_filename = (char *)(memory_next_filename + offset);
             new_dentry =
                (struct dentry_list *)(malloc(sizeof(struct dentry_list )));
             new_dentry->previous = last_dentry;
             last_dentry = new_dentry;
             strncpy(new_dentry->name, next_filename, MAX_LENGTH-1);
        }
        file_name[0] = '\0';
        if(!(strstr(type, "SOCK")))
        {
             while(last_dentry != NULL)
             {
                  struct dentry_list *old_dentry = last_dentry;
                  if(strncmp(last_dentry->name, "/", MAX_LENGTH-1))
                  {
                       strcat(file_name, last_dentry->name);
                       strcat(file_name, "/");
                  }
                  else strcat(file_name, "/");
                  last_dentry = last_dentry->previous;
                  free(old_dentry);
             }
             file_name[strlen(file_name) -1] = '\0';
        }
        else strcat(file_name, "socket");
        free(last_dentry);
        if(!strcmp(type,"REG"))
        {
             /*queste informazioni sono utilizzate dal manager */
             printf("%d#%lu#%s\n", pid, node, file_name);
        }
        munmap(memory_next_file, XA_PAGE_SIZE);
        munmap(memory_dentry, XA_PAGE_SIZE);
        munmap(memory_next_filename, XA_PAGE_SIZE);
        ++fd;
   }
   munmap(memory_file, XA_PAGE_SIZE);
   munmap(memory_fdt, XA_PAGE_SIZE);
```

```
          munmap(memory_fd, XA_PAGE_SIZE);
          munmap(memory, XA_PAGE_SIZE);
          memory = xa_access_virtual_address(&xai, next_addr, &offset);
     }

     xc_domain_unpause(xai.xc_handle, dom); /* esecuzione del dominio riavviata */

error_exit:
     xa_destroy(&xai);
     if (memory) munmap(memory, XA_PAGE_SIZE);

     return 0;
}
```

```
/*
 * xen_vmi.h
 *
 * Dichiarazioni utili per il programma xen_check_dom
 */
#ifndef XEN_VMI_HEADER
#define XEN_VMI_HEADER

#define MODULE_NAME_LEN (64 - sizeof(unsigned long))
#define DIGEST_SIZE 20
#define XC_PAGE_SHIFT 12
#define IDT_ENTRIES 256
#define XC_PAGE_SIZE (1UL << XC_PAGE_SHIFT)
#define XA_PAGE_SIZE XC_PAGE_SIZE

#define if_free(p) if(p!=NULL) free(p)

typedef unsigned int uint32_t;
typedef unsigned char uint8_t;
typedef unsigned short int uint16_t;

/*per memorizzare le informazioni di controllo sui moduli */
struct mod_info
{
     char mod_name[MODULE_NAME_LEN];
     unsigned char *mod_text;
     unsigned long mod_size;
     int license;
};

/*per memorizzare i digest */
struct digest_info
{
     char name[MODULE_NAME_LEN];
     unsigned char digest[DIGEST_SIZE];
};

/*trap_table di Xen */
typedef struct trap_info
{
     uint8_t       vector;  /* exception vector                              */
     uint8_t       flags;   /* 0-3: privilege level; 4: clear event enable?  */
     uint16_t      cs;      /* code selector                                 */
     unsigned long address; /* code offset                                   */
} trap_info_t;

/*dichiarazione delle funzioni utilizzate */
void calculate_hash(unsigned char **msg, size_t len,
                     int max, unsigned char **md_value, unsigned int *md_len);
int get_modules_infos(struct mod_info ***info_ptr,int *num_modules,uint32_t dom);
int get_syscall_table(uint32_t **sys_table, int *max, uint32_t dom);
int get_kernel_text(unsigned char ***text_page, int *page_num, uint32_t dom);
int get_idt_table(unsigned char ***idt_table, int *idt_entries, uint32_t dom);
```

```
int get_process_list(uint32_t dom);
int get_process_infos(uint32_t dom);
int if_promisc(uint32_t dom);
int list_open_files(uint32_t dom);

#endif
```

```
/*
 * xen_hash.c
 *
 * Definisce la funzione per calcolare l'hash delle informazioni
 */
#include <openssl/evp.h>      /* header della libreria
#include <openssl/crypto.h>      OpenSSL */
#include "xen_vmi.h"


/*restituisce l'hash di msg, calcolato utilizzando l'algoritmo SHA1.
  msg=array di puntatori, es. a pagine
  len=lunghezza singola pagina
  max=numero pagine */
void calculate_hash(unsigned char **msg, size_t len,
                      int max, unsigned char **md_value, unsigned int *md_len)
{
     EVP_MD_CTX mdctx;
     const EVP_MD *md = EVP_sha1();
     int i;

     *md_value = (unsigned char*)malloc(sizeof(unsigned char) * EVP_MAX_MD_SIZE);

     OpenSSL_add_all_digests();

     if(!md)
     {
         printf("Unknown message digest \n");
         exit(1);
     }

     EVP_MD_CTX_init(&mdctx);
     EVP_DigestInit_ex(&mdctx, md, NULL);
     for(i=0; i<max; i++)
     {
         EVP_DigestUpdate(&mdctx, msg[i], len);
     }

     EVP_DigestFinal_ex(&mdctx, *md_value, md_len);
     EVP_MD_CTX_cleanup(&mdctx);

}
```

```
/*
 * xen_check_dom.c
 *
 * Programma che crea la lista di controllo ed invoca le funzioni di introspezione
 * effettuando i controlli di consistenza
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include "xen_vmi.h"
```

```
/*dichiarazione della lista di controllo */
int list_size;
struct digest_info **digestList;

/*crea la lista di controllo */
int init_vmi(uint32_t dom)
{
     int i=0, j=0;
     unsigned char **msg;
     unsigned char *md_value;
     unsigned int md_len;

     uint32_t *sys_table;
     int sys_num;

     unsigned char **kernel_pages;
     int kernel_page_num;

     unsigned char **idt_table;

     struct mod_info **list = NULL;
     int n_modules = 0;

     struct digest_info *elem = NULL;
     unsigned char *tmp = NULL;


     printf("***init check list***\n");

     get_modules_infos(&list,&n_modules,dom);
     list_size = n_modules + 3;
     digestList =
       (struct digest_info **)(malloc(list_size * sizeof(struct digest_info *)));

     /*get the digest of all lodable kernel module*/
     for(i=0; i<n_modules; i++)
     {
         tmp = list[i]->mod_text;
         calculate_hash(&tmp, list[i]->mod_size, 1, &md_value, &md_len);
         elem = malloc(sizeof(struct digest_info));
         strcpy(elem->name, list[i]->mod_name);
         memcpy(elem->digest, md_value, DIGEST_SIZE);
         (digestList)[i] = elem;
         free(tmp);
         if_free(list[i]);
         free(md_value);
     }
     if_free(list);

     /*get the digest of system call table*/
     get_syscall_table(&sys_table, &sys_num, dom);
     msg = (unsigned char **)(malloc(sizeof(unsigned char *)));
     msg[0] = (unsigned char*)sys_table;
     calculate_hash(msg, sizeof(uint32_t) * sys_num, 1, &md_value, &md_len);
     elem = malloc(sizeof(struct digest_info));
     sprintf(elem->name, "sys_call_table");
     memcpy(elem->digest, md_value, DIGEST_SIZE);
     (digestList)[i] = elem;
     free(sys_table);
     free(msg);
     free(md_value);
     i=i+1;

     /*get the digest of kernel text area*/
     get_kernel_text(&kernel_pages, &kernel_page_num, dom);
     calculate_hash(kernel_pages, XA_PAGE_SIZE,
                     kernel_page_num, &md_value, &md_len);
     elem = malloc(sizeof(struct digest_info));
     sprintf(elem->name, "kernel_text_area");
     memcpy(elem->digest, md_value, DIGEST_SIZE);
```

```
     (digestList)[i] = elem;
     for(j=0; j< kernel_page_num; j++) free(kernel_pages[j]);
     free(kernel_pages);
     free(md_value);
     i=i+1;

     /*get the digest of interrupt table*/
     get_idt_table(&idt_table, dom);
     calculate_hash(idt_table, sizeof(trap_info_t),
                    IDT_ENTRIES, &md_value, &md_len);
     elem = malloc(sizeof(struct digest_info));
     sprintf(elem->name, "interrupt_table");
     memcpy(elem->digest, md_value, DIGEST_SIZE);
     (digestList)[i] = elem;
     for(j=0; j<IDT_ENTRIES; j++) free(idt_table[j]);
     free(idt_table);

#if 0
     printf("# List all check list info about dom %d\n", dom);
     for(i=0;i<list_size;i++) {
         printf("%d. [%s] digest: ", (i+1), (digestList)[i]->name);
         for(j=0; j<DIGEST_SIZE; j++) printf("%02x", (digestList)[i]->digest[j]);
         printf("\n");
     }
     printf("\n");
#endif

     return 0;
}

/*elimina la lista di controllo */
void clean_data(int sig)
{
     int i = 0;

     printf("***clean check list***\n");
     for(i=0; i<list_size; i++) if_free(digestList[i]);
     if_free(digestList);
     exit(0);
}

/*controlla se l'hash di un'informazione è corretto, accedendo alla lista di
  controllo.
  check_name=nome che identifica l'informazione nella lista di controllo
  check_digest=digest che deve essere controllato */
int check_hash(char *check_name, unsigned char *check_digest)
{
     int i = 0;

     for(i=0; i<list_size; i++)
     {
         if(!strcmp(digestList[i]->name,check_name))
         {
              if(!memcmp((digestList)[i]->digest,check_digest,DIGEST_SIZE))
                  return 1;
              else return 0;
         }
     }
     return -1;
}


int main(int argc, char **argv)
{
     uint32_t dom = 0;
     int opt, time = 30;
     int init = 1;
     int i=0, j=0;
     unsigned char **msg;
     unsigned char *md_value;
```

```
unsigned int md_len;

uint32_t *sys_table;
int sys_num;

unsigned char **kernel_pages;
int kernel_page_num;

unsigned char **idt_table;

struct mod_info **list = NULL;
int size = 0;

struct sigaction action;
char tmp[DIGEST_SIZE];

/*Parameters*/
int modules = 1;
int syscall = 1;
int kernel_text = 1;
int idt = 1;

if(argc < 2)
{
    printf("usage:\n \
         xen_check_dom <DomId>\n \
         xen_check_dom <DomId> -t <time> \
         xen_check_dom <DomId> <Options: [-l][-p][-m][-s][-k][-i]>\n");
    return 0;
}
dom = atoi(argv[1]); /*identificatore del dominio monitorato */
/*se non sono specificati altri parametri vengono effettuati tutti i
  controlli di consistenza, con un intervallo di attesa di 30 secondi */
if(argc > 2)
{
    opt=getopt(argc,argv,"tlpmski");
    switch(opt) {
    /*intervallo di attesa in secondi */
    case 't' : time = atoi(argv[3]);
         break;
    /*introspezione sulla lista dei file aperti */
    case 'l' : get_open_files(dom);
         return 0;
    /*introspezione sulla lista dei processi attivi */
    case 'p' : get_process_list(dom);
         return 0;
    /*introspezione sulla lista dei moduli caricati nel kernel*/
    case 'm' : syscall = kernel_text = idt = 0;
         break;
    /*introspezione sulla tabella delle System Call */
    case 's' : modules = kernel_text = idt = 0;
         break;
    /*introspezione sulla text area del kernel */
    case 'k' : modules = syscall = idt = 0;
         break;
    /*introspezione sulla tabella delle interruzioni */
    case 'i' : modules = syscall = kernel_text = 0;
         break;

    default  : printf("unknow options –
                        Options: [-t][-l][-p][-m][-s][-k][-i]\n");
         return 0;
    }
}

action.sa_handler=clean_data;
/*segnali di terminazione */
sigaction(SIGINT,&action,NULL);
sigaction(SIGTERM,&action,NULL);
```

```
while(1) /*il programma rimane in esecuzione fino a quando non viene
{          terminato dall'amministratore */
    /*quando il programma viene eseguito crea la lista di controllo */
    if(init) init=init_vmi(dom);

    /*esecuzione dei controlli di consistenza */
    printf("***xen vmi run***\n");

    if(modules)
    {
         get_modules_infos(&list,&size,dom);
         /*verifica delle informazioni sui moduli del kernel */
         for(i=0; i<size; i++)
         {
             if(!(list[i]->license))
         printf("Module [%s] is not GPL-compatible\n", list[i]->mod_name);
             else {
                 calculate_hash(&(list[i]->mod_text),
                                list[i]->mod_size, 1, &md_value, &md_len);
                 /*print_hex(list[i]->mod_text, list[i]->mod_size);*/
                 printf("Module [%s] digest is:\n", list[i]->mod_name);
                 /*for(j=0; j<md_len; j++) printf("%02x", md_value[j]);*/
                 j = check_hash(list[i]->mod_name,md_value);
                 if(j > 0) printf("\tcorrect\n");
                 else if(!j) printf("\tNOT correct\n");
                 else printf("\tNOT FOUND\n");
             }
             if_free(list[i]->mod_text);
             if_free(list[i]);
             free(md_value);
         }
         if_free(list);
    }

    if(syscall)
    {
         get_sys_call_table(&sys_table, &sys_num, dom);
         /*verifica delle informazioni sulla System Call Table */
         msg = (unsigned char **)(malloc(sizeof(unsigned char *)));
         msg[0] = (unsigned char*)sys_table;
         calculate_hash(msg, sizeof(uint32_t) * sys_num, 1,
                                    &md_value, &md_len);
         printf("System call table digest is:\n");
         /*for(j=0; j<md_len; j++) printf("%02x", md_value[j]);*/
         sprintf(tmp, "sys_call_table");
         if(check_hash(tmp,md_value)) printf("\tcorrect\n");
         else printf("\tNOT correct\n");

         free(sys_table);
         free(msg);
         free(md_value);
    }

    if(kernel_text)
    {
         get_kernel_text(&kernel_pages, &kernel_page_num, dom);
         /*verifica delle informazioni sulla text area del kernel */
         calculate_hash(kernel_pages, XA_PAGE_SIZE,
                        kernel_page_num, &md_value, &md_len);
         printf("Kernel text area digest is: \n");
         /*for(j=0; j<md_len; j++) printf("%02x", md_value[j]);*/
         sprintf(tmp, "kernel_text_area");
         if(check_hash(tmp,md_value)) printf("\tcorrect\n");
         else printf("\tNOT correct\n");

         for(i=0; i<kernel_page_num; i++) free(kernel_pages[i]);
         free(kernel_pages);
         free(md_value);
    }
```

```
          if(idt)
          {
               get_idt_table(&idt_table, dom);
               /* verifica delle informazioni sulla tabella delle interruzioni */
               calculate_hash(idt_table, sizeof(trap_info_t),
                                    IDT_ENTRIES, &md_value, &md_len);
               printf("Idt table digest is: \n");
               /*for(j=0; j<md_len; j++) printf("%02x", md_value[j]);*/
               sprintf(tmp, "interrupt_table");
               if(check_hash(tmp,md_value)) printf("\tcorrect\n");
               else printf("\tNOT correct\n");

               for(i=0; i<IDT_ENTRIES; i++) free(idt_table[i]);
               free(idt_table);

          }

          /*introspezione sulle interfacce di rete */
          print_promisc_if(dom);

          printf("\n");
          sleep(time); /*fine iterazione, attesa... */

     }

     return 0;

}
```

```
/*
 * MAKEFILE
 *
 * Si assume che il kenel del dominio guest sia localizzato in
 * /xen/xen-3.0.2/linux-2.6.16-xenU
 */
PREFIX=/xen/xen-3.0.2/linux-2.6.16-xenU/include
HEADERS=-I$(PREFIX)/ -I$(PREFIX)/asm-i386/mach-generic/
         -I$(PREFIX)/asm-i386/mach-default/
LIBS=-lxenctrl -lxenaccess -lss -lcrypto
LIBDIR=/usr/local/lib/
CFLAGS=-Wall
CFLAGS_KERNEL= -D __KERNEL__  -D"KBUILD_BASENAME=KBUILD_STR(xen_vmi)"
               -D"KBUILD_MODNAME=KBUILD_STR(xen_vmi)"
CC = gcc
LINK=-Wl,--rpath -Wl,$(LIBDIR)
OBJS=xen_vmi.o xen_hash.o

all:xen_vmi xen_hash xen_check_dom

xen_vmi: Makefile xen_vmi.c xen_vmi_private.h
       $(CC) $(HEADERS) -D"KBUILD_STR(s)=#s" $(CFLAGS) $(CFLAGS_KERNEL) -c $@.c
xen_hash: Makefile xen_hash.c xen_vmi.h
       $(CC) $(CFLAGS) -c $@.c

xen_check_dom: Makefile xen_check_dom.c xen_vmi.h $(OBJS)
       $(CC) $(CFLAGS) $(LINK) $(LIBS) -o $@ $@.c $(OBJS)

clean:
       -rm -f *.o *~
```

B. Moduli del kernel

```
/*
 * antisniff.c
 *
 * Modulo che controlla se le interfacce di rete sono in modalita promiscua */
 */
#include <linux/module.h>            /* Needed by all modules */
#include <linux/kernel.h>            /* Needed for KERN_INFO */
#include <linux/init.h>              /* Needed for the macros */
#include <linux/moduleparam.h>       /* Needed to passing command line arguments*/
#include <linux/netdevice.h>         /* Needed for struct net_device */
#include <linux/sched.h>             /* We need to put ourselves to sleep and
                                        wake up later */
#include <linux/workqueue.h>         /* We scheduale tasks here */
#include <linux/string.h>

/*parametro=intervallo di attesa in secondi */
static int time = 30;
module_param(time, int, 0644);

static void checkPromisc(void *);

static int die = 0; /* set this to 1 for shutdown */
/*The work queue structure for this task */
static struct workqueue_struct *my_workqueue;
static struct work_struct Task;
/*Task hold a pointer to the function */
static DECLARE_WORK(Task, checkPromisc, NULL);

/*This function will be called once for every timer interrupt */
static void checkPromisc(void *irrelevant) {

        struct net_device *dev;

        /*controllo sulla lista dei dispositivi di rete memorizzata all'indirizzo
          dev_base */
        for(dev = dev_base ; dev; dev = dev->next) {
                if(strstr(dev->name,"eth")) {
                        if((dev->flags & IFF_PROMISC) ||
                                (dev->gflags && IFF_PROMISC) || dev->promiscuity) {
                                printk(KERN_INFO
                                        "[antisniff] %s: PROMISC\n", dev->name);
                        }
                        else printk(KERN_INFO
                                        "[antisniff] %s: not promisc\n", dev->name);
                }
        }
        /*If cleanup wants us to die */
         if(die == 0) queue_delayed_work(my_workqueue, &Task, time*100);

}

/*funzione invocata quando il modulo viene caricato nel kernel */
static int __init antisniff(void)
{
        printk(KERN_INFO "antisniff init.\n");
        /*Put the task in the work_timer task queue,
          so it will be executed at next timer interrupt */
        my_workqueue = create_workqueue("antisniff");
        queue_delayed_work(my_workqueue, &Task, time*100);

        return 0;
}

/*funzione invocata quando il modulo viene rimosso dal kernel */
static void __exit cleanup(void)
{
        die = 1;                            /*keep function from queueing itself*/
```

```
        cancel_delayed_work(&Task);          /*no "new ones"*/
        flush_workqueue(my_workqueue);       /*wait till all "old ones" finished*/
        destroy_workqueue(my_workqueue);
        /*This is necessary, because otherwise we'll deallocate the memory
          holding function and Task while work_timer still references them */
        printk(KERN_INFO "antisniff exit.\n");

}

MODULE_DESCRIPTION("Detect if network device is in promiscus mode");
MODULE_LICENSE("GPL");

module_init(antisniff);
module_exit(cleanup);
```

```
/*
 * check_proc.c
 *
 * Modulo che controlla la text area dei processi attivi
 */
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/workqueue.h>
/*per accedere allo spazio di indirizzamento dei processo */
#include <linux/ptrace.h>
#include <linux/mm.h>
#include <linux/highmem.h>
/*per calcolare gli hash con le funzini crypto_digest */
#include <linux/crypto.h>
#include <asm/scatterlist.h>
#include <linux/pagemap.h>

#define MAX_PROC_NUM     64
#define SHA1_DIGEST_SIZE 20

/*per memorizzare i digest */
struct digest
{
        char name[TASK_COMM_LEN];
        unsigned char value[SHA1_DIGEST_SIZE];
        struct digest *next;
};

/*parametro=intervallo di attesa in secondi */
static int time = 60;
module_param(time, int, 0644);

static void checkProcess(void *);

static int die = 0; /* set this to 1 for shutdown */
/*The work queue structure for this task */
static struct workqueue_struct *my_workqueue;
static struct work_struct Task;
/*Task hold a pointer to the function */
static DECLARE_WORK(Task, checkProcess, NULL);
/*Process .text digest list*/
static int init_count = 0;
struct digest *list = NULL;

/*per accedere allo spazio di indirizzamanto di un processo */
int access_process_vm(struct task_struct *tsk,
                      unsigned long addr, void *buf, int len, int write)
{
        struct mm_struct *mm;
        struct vm_area_struct *vma;
        struct page *page;
        void *old_buf = buf;
```

```
        mm = get_task_mm(tsk);
        if(!mm) return 0;

        down_read(&mm->mmap_sem);
        /* ignore errors, just check how much was sucessfully transfered */
        while(len) {
                int bytes, ret, offset;
                void *maddr;

                ret = get_user_pages(tsk, mm, addr, 1, write, 1, &page, &vma);
                if(ret <= 0) break;

                bytes = len;
                offset = addr & (PAGE_SIZE-1);
                if(bytes > PAGE_SIZE-offset) bytes = PAGE_SIZE-offset;

                maddr = kmap(page);
                if(write) {
                    copy_to_user_page(vma, page, addr, maddr + offset, buf, bytes);
                    set_page_dirty_lock(page);
                }
                else
                    copy_from_user_page(vma, page, addr, buf, maddr+offset, bytes);
                kunmap(page);
                page_cache_release(page);
                len -= bytes;
                buf += bytes;
                addr += bytes;
        }
        up_read(&mm->mmap_sem);
        mmput(mm);

        return buf - old_buf;
}

/*restituisce la zona di memoria richiesta e associata al processo specificato */
int ptrace_kreaddata(struct task_struct *tsk,
                        unsigned long src, char __user *dst, int len)
{
        int copied = 0;

        while(len > 0) {
                char buf[128];
                int this_len, retval;

                this_len = (len > sizeof(buf)) ? sizeof(buf) : len;
                retval = access_process_vm(tsk, src, buf, this_len, 0);
                if(!retval) {
                    if(copied) break;
                    return -EIO;
                }
                if(memcpy(dst, buf, retval) == NULL) return -EFAULT;
                copied += retval;
                src += retval;
                dst += retval;
                len -= retval;
        }
        return copied;
}

/*controlla se l'hash relativo al processo specificato è gia stato inserito nella
  lista di controllo */
int repeat(char *name, int len)
{
     int i = 0;
     struct digest *tmp = list;

     for(i=0;i<len;i++) {
       if(!strcmp(tmp->name,name)) return 1;
```

```
        tmp = tmp->next;
      }
      return 0;
}

/*crea la lista di controllo, contenente l'hash di tutti i processi eseguibili */
int create_digest_list(void)
{
      struct crypto_tfm *tfm;
      struct scatterlist sg[1];
      char *dst;
      struct task_struct *task = next_task(&init_task);
      int count = 0;
      struct digest *last = list;


      tfm = crypto_alloc_tfm("sha1", CRYPTO_TFM_REQ_MAY_SLEEP);
      if(tfm == NULL) return 0;
      dst = (char *)__get_free_page(GFP_KERNEL);

      for_each_process(task) {
        if(task->mm != NULL && !repeat(task->comm,count))
        {
           unsigned long addr;
           unsigned long end;
           struct digest *tmp;

           addr = task->mm->start_code;
           end = task->mm->end_code;
           crypto_digest_init(tfm);
           while((end-addr) > PAGE_SIZE) {
                sg[0].page = virt_to_page(dst);
                sg[0].offset = 0;
                sg[0].length = PAGE_SIZE;
                ptrace_kreaddata(task, addr, (char __user *)dst, PAGE_SIZE);
                crypto_digest_update(tfm, sg, 1);
                addr+=PAGE_SIZE;
           }
           sg[0].page = virt_to_page(dst);
           sg[0].offset = 0;
           sg[0].length = (end-addr);
           ptrace_kreaddata(task, addr, (char __user *)dst, (end-addr));
           crypto_digest_update(tfm, sg, 1);
           /*add new element to digest list*/
           if(list == NULL)
              tmp=list=(struct digest *)kmalloc(sizeof(struct digest), GFP_KERNEL);
           else tmp = last->next =
                       (struct digest *)kmalloc(sizeof(struct digest), GFP_KERNEL);
           last = tmp;
           strncpy(tmp->name, task->comm, TASK_COMM_LEN);
           crypto_digest_final(tfm, tmp->value);
           /*printk(KERN_INFO "%d.[%s] digest add\n", count+1, tmp->name);*/
           count++;
        }
      }
      crypto_free_tfm(tfm);
      free_page((unsigned long)dst);

      return count;
}

/*controlla se l'hash associato ad un processo è corretto, accedendo alla lista di
  controllo */
int compare_digest(char *name, unsigned char *digest)
{
      int i = 0;
      struct digest *tmp = list;

      for(i=0;i<init_count;i++) {
        if(!strcmp(tmp->name,name)) {
```

```
            if(!memcmp(tmp->value,digest,SHA1_DIGEST_SIZE)) return 1;
            else return 0;
          }
          tmp = tmp->next;
        }

        return -1;
}

/*elimina la lista di controllo */
void clean_digest_list(void)
{
        int i = 0;
        struct digest *tmp = list;

        for(i=0;i<init_count;i++)
        {
          list = tmp->next;
          kfree(tmp);
          tmp = list;
        }

}

/*This function will be called once for every timer interrupt */
static void checkProcess(void *irrelevant)
{
        struct crypto_tfm *tfm;
        struct scatterlist sg[1];
        char *dst;
        struct task_struct *task = next_task(&init_task);
        unsigned char hash[SHA1_DIGEST_SIZE];
        int tmp = 0;

        if(!init_count) {
          printk(KERN_INFO "Failed to create process .text digest list\n");
          return;
        }

        tfm = crypto_alloc_tfm("sha1", CRYPTO_TFM_REQ_MAY_SLEEP);
        if(tfm == NULL) {
          printk(KERN_INFO "Failed to load transform for SHA1\n");
          return;
        }
        dst = (char *)__get_free_page(GFP_KERNEL);

        /*ciclo che calcola l'hash della text area di tutti i processi attivi */
        for_each_process(task) {
          if(task->mm != NULL)
          {
             unsigned long addr;
             unsigned long end;

             addr = task->mm->start_code;
             end = task->mm->end_code;

             /*procedura di recupero della text area e calcolo dell'hash */
             crypto_digest_init(tfm);
             while((end-addr) > PAGE_SIZE) {
                  sg[0].page = virt_to_page(dst);
                  sg[0].offset = 0;
                  sg[0].length = PAGE_SIZE;
                  ptrace_kreaddata(task, addr, (char __user *)dst, PAGE_SIZE);
                  crypto_digest_update(tfm, sg, 1);
                  addr+=PAGE_SIZE;
             }
             sg[0].page = virt_to_page(dst);
             sg[0].offset = 0;
             sg[0].length = (end-addr);
             ptrace_kreaddata(task, addr, (char __user *)dst, (end-addr));
```

```
            crypto_digest_update(tfm, sg, 1);
            crypto_digest_final(tfm, hash);
            printk(KERN_INFO "[%s] text area digest is: ", task->comm);
            /*for(tmp=0; tmp<SHA1_DIGEST_SIZE; tmp++)
                printk(KERN_INFO "%02x", hash[tmp]);*/
            /*check if the hash value is correct*/
            tmp = compare_digest(task->comm, hash);
            if(tmp > 0) printk(KERN_INFO "\tcorrect\n");
            else if(!tmp) printk(KERN_INFO "\tNOT correct\n");
            else printk(KERN_INFO "\tNOT FOUND\n");
          }
      }
      crypto_free_tfm(tfm);
      free_page((unsigned long)dst);
      /*If cleanup wants us to die */
      if(die == 0) queue_delayed_work(my_workqueue, &Task, time_wait*100);

}

/*funzione invocata quando il modulo viene caricato nel kernel */
static int __init start(void)
{
      printk(KERN_INFO "***check_proc init***\n");
      /*init process .text digest list*/
      if(!init_count) init_count = create_digest_list();
      /*Put the task in the work_timer task queue,
        so it will be executed at next timer interrupt */
      my_workqueue = create_workqueue("checkProcess");
      queue_delayed_work(my_workqueue, &Task, time_wait*100);

      return 0;
}

/*funzione invocata quando il modulo viene rimosso dal kernel */
static void __exit cleanup(void)
{
        die = 1;                              /*keep function from queueing itself*/
        cancel_delayed_work(&Task);           /*no "new ones"*/
        flush_workqueue(my_workqueue);        /*wait till all "old ones" finished*/
        destroy_workqueue(my_workqueue);
        /*This is necessary, because otherwise we'll deallocate the memory
          holding function and Task while work_timer still references them */
        clean_digest_list(); /*clean process .text digest list*/
        printk(KERN_INFO "***check_proc exit***\n");

}

MODULE_DESCRIPTION("Check process text area");
MODULE_LICENSE("GPL");

module_init(start);
module_exit(cleanup);



/*
 * MAKEFILE
 */
obj-m += antisniff.o
obj-m += check_proc.o
all:
        make -C /lib/modules/2.6.16-xenU/build M=$(PWD) modules
clean:
        make -C /lib/modules/2.6.16-xenU/build M=$(PWD) clean
```

C. Agent-Manager

```
#
# agentActions.py
#
# Definizione delle funzioni utilizzate dagli agenti per svolgere le loro azioni
#
import os
import string
import socket
import sys, traceback

#per chiudere la connessione con il manager
def closeApp(conn):
    msg = 'OK'
    while (len(msg) > 0):
        try:
            ns = conn.send(msg)
        except:
            conn.close()
            raise
        msg = msg[ns:]
    data = conn.recv(16)
    if not data:
        print 'closeApp connection closed'
        conn.close()

#per parametrizzare un insieme di informazioni di configurazione
#se si deve monitorare un solo dominio non è necessario utilizzare questa funzione
def receiveConf(conn, file):
    msg = 'OK'
    while (len(msg) > 0):
        try:
            ns = conn.send(msg)
        except:
            conn.close()
            raise
        msg = msg[ns:]
    try:
        socketFile = conn.makefile('r', 0)
        lines = socketFile.readlines()
    except:
        socketFile.close()
        conn.close()
        raise
    output = open(file, 'w')
    output.writelines(lines)
    socketFile.close()
    output.close()
    conn.close()

#funzione wrapper di ps
def showProc(conn):
    command = 'ps -e -o pid -o uid -o command'
    #acquisizione delle informazioni
    output = os.popen(command)
    lines = output.readlines()
    lines = lines[1:len(lines)-1]
    returnLines = []
    #formattazione dell'output
    for line in lines:
        fields = line.split()
        pid = fields[0]
        uid = fields[1]
        command = fields[2]
        command= command.replace('[','').replace(']','')
        index = command.rfind('/')
        if index > 0 and not command[index+1] == '0':
            command = command[index+1:]
```

```
        if command[0] == '-':
            command = command[1:]
        if command[len(command)-1] == ':':
            command = command[:len(command) - 1]
        line = pid+" "+uid+" "+command+"\n";
        returnLines.append(line)
    try:
        try:
            socketFile = conn.makefile('w', 0)
            #le informazioni ottenute vengono inviate al manager
            socketFile.writelines(returnLines)
        except:
            raise
    finally:
        socketFile.close()
        conn.close()

#funzione wrapper di lsof
def execLSOF(conn):
    """
    COMMAND,PID,UID,TYPE,SIZE,NODE,NAME: lsof -F cputsin0
    """
    command = "lsof -F tin0"
    #acquisizione delle informazioni
    output = os.popen(command)
    lines = output.readlines()
    lsof = []
    #formattazione dell'output
    for line in lines:
        line = line.replace('\n','')
        line = line.replace('\x00','')
       if line[0] == 'p':
            pid = line[1:]
       elif line.find("REG") > 0:
           types, tmp, rest = line.partition('i')
            inode, tmp, name = rest.partition('n')
            format = pid+"#"+inode+"#"+name+"\n"
            lsof.append(format)
    try:
        try:
            socketFile = conn.makefile('w', 0)
            #le informazioni ottenute vengono inviate al manager
            socketFile.writelines(lsof)
        except:
            raise
    finally:
       socketFile.close()
        conn.close()




#
# agent_xen_vmi.py
#
# Programma che esegue gli agenti
#
import logging
import socket
import os
import agentActions

#nome del dominio monitorato
vmName = 'moduleDom'

class AgentXenVMI:

     def __init__(self, interval):
       self.myName = self.__class__.__name__
```

```
        #intervallo di attesa, specificato per i moduli del kernel da eseguire
        self.interval = interval
        #caricamento del modulo antisniff
        commands.getstatusoutput("modprobe antisniff time=" + self.interval)
        #caricamento del modulo check_proc
        commands.getstatusoutput("modprobe check_proc time=" + self.interval)
        print 'DEBUG, agentPS started\n'
        print 'DEBUG, agentLSOF started\n'

    def start(self):

        host = "192.168.1.5"
        port = 3434
        #apertura del socket di ascolto per le richieste del manager
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        print host, port, vmName
        self.sock.bind((host, port))
        self.sock.listen(5)
        abort = 0

        print 'agent xen vmi started'
        while not abort:
            #in attesa delle richieste del manager
            conn, addr = self.sock.accept()
            data = conn.recv(16)
            print data
            if data == 'CONFIG':
                try:
                   print "not used"
                   #agentActions.receiveConf(conn, self.loadconf.confFile)
                except:
                    #do something
                    pass
            elif data == 'PS':
                try:
                    #richiesta della lista dei processi attivi
                    agentActions.showProc(conn)
                except:
                    #do something
                    pass
            elif data == 'LSOF':
                try:
                    #richiesta della lista dei file aperti
                    agentActions.execLSOF(conn)
                except:
                    #do something
                    pass
            elif data == 'CLOSEAPP':
                try:
                    #richiesta di terminazione
                    abort = 1;
                    agentActions.closeApp(conn)
                except:
                    #do something
                    pass
            else:
                print 'Unknown request'

        self.sock.close()
        print 'DEBUG', self.myName + ' stopped\n'

if __name__ == '__main__':
    #avvio del thread
    c=AgentXenVMI(sys.argv[1])
    c.start()
```

```
#
# managerActions.py
#
# Definizione delle funzioni utilizzate dal manager per svolgere le sue azioni
#
import socket
import sys, traceback

timeout = 20

#per chiudere la connessione con il manager
def closeApp(host, port):
    """
    Send the host the close application command
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
    except:
        raise
    try:
        s.connect((host, port))
    except:
        raise
    msg = 'CLOSEAPP'
    while (len(msg) > 0):
        try:
            ns = s.send(msg)
        except:
            s.close()
            raise
        msg = msg[ns:]
    data = s.recv(16)
    if data == 'OK':
        print 'interrupt'
        s.close()
    else:
        s.close()

#per parametrizzare un insieme di informazioni di configurazione
#se si deve monitorare un solo dominio non è necessario utilizzare questa funzione
def sendConf(host, port, file):
    """
    Send the configuration file to the host
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
    except:
        raise
    try:
        s.connect((host, port))
    except:
        raise
    msg = 'CONFIG'
    while (len(msg) > 0):
        try:
            ns = s.send(msg)
        except:
            s.close()
            raise
        msg = msg[ns:]
    while 1:
        data = s.recv(16)
        if not data: break
        if data == 'OK':
            input = open(file, 'r')
            lines = input.readlines()
            input.close()
            try:
```

```
                try:
                    socketFile = s.makefile('w', 0)
                    socketFile.writelines(lines)
                except:
                    raise
            finally:
                socketFile.close()
                s.close()
        else:
            pass
    s.close()

#per ricevere la lista dei processi attivi inviata dall'agent
def showProc(host, port):
    """
    Return a list of the processes currently running on host
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
    except:
        raise
    try:
        s.connect((host, port))
    except:
        raise
    msg = 'PS'
    while (len(msg) > 0):
        try:
            ns = s.send(msg)
        except:
            raise
        msg = msg[ns:]
    try:
        try:
            socketFile = s.makefile('r', 0)
        except:
            raise
        else:
            ps = []
            for line in socketFile:
                ps.append(line)
            return ps
    finally:
        socketFile.close()
        s.close()

#per ricevere la lista dei file aperti inviata dall'agent
def execLSOF(host, port):
    """
    Return a list of open files on host
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
    except:
        raise
    try:
        s.connect((host, port))
    except:
        raise
    msg = 'LSOF'
    while (len(msg) > 0):
        try:
            ns = s.send(msg)
        except:
            raise
        msg = msg[ns:]
    try:
        try:
```

```
            socketFile = s.makefile('r', 0)
        except:
            raise
        else:
            lsof = []
            for line in socketFile:
                lsof.append(line)
            return lsof
    finally:
        socketFile.close()
        s.close()



#
# manager_xen_vmi.py
#
# Programma che esegue il manager
#
import time
import sys
import socket
import os
import commands
import managerActions
from threading import *
from thread import *


class ManagerXenVmi(Thread):

    def __init__(self, DomId):
        Thread.__init__(self)
        self.myName = self.__class__.__name__
        self.DomId = DomId
        print 'DEBUG', self.myName + ' started\n'
        self._finished = Event()
        self._interval = 30.0
        #attivazione dei controlli di consistenza eseguiti da xen_check_dom
        commands.getstatusoutput("./xen_check_dom " +
                                    self.DomId + " -t " + self._interval)
        self._ps = True
        self._lsof = True


    def run(self):
        host = "192.168.1.5"
        port = 3434
        try:
            while 1:
               if self._lsof:
                    result = {}
                    client = {}
                    #invio della richiesta della lista dei file aperti all'agent
                    lsof = managerActions.execLSOF(host,port)
                    #esecuzione della funzione di introspezione list_open_files()
                    output = os.popen("./xen_check_dom " + self.DomId + " -l")
                    lines = output.readlines()
                    #elaborazione delle due liste, per un confronto piu semplice
                    for line in lines:
                      pid, inode, name = line.split('#')
                      pid = pid.replace('\x00','')
                      if not result.has_key(pid): result[pid] = {}
                      result[pid][inode] = name.replace('\n','')

                    for line in lsof:
                      pid, inode, name = line.split('#')
                      if not client.has_key(pid): client[pid] = {}
                      client[pid][inode] = name.replace('\n','')
```

```
                    #confronto tra le due liste
                    if not cmp(result,client) > 0: print 'LSOF OK'
                    else:
                        print "hidden files:"
                        key = result.keys()
                        key.sort()
                        for k in key:
                            if cmp(result[k],client[k]):
                                value = result[k].keys()
                                for val in value:
                                    if not client[k].has_key(val):
                                        print k, val, result[k][val]

                if self._ps:
                    result = {}
                    client = {}
                    #invio della richiesta per lista dei processi attivi all'agent
                    ps = managerActions.showProc(host, port)
                    #esecuzione della funzione di introspezione get_process_list()
                    output = os.popen("./xen_check_dom " + self.DomId + " -p")
                    lines = output.readlines()
                    #elaborazione delle due liste, per un confronto piu semplice
                    for line in lines:
                        pid, uid, command = line.split()
                        result[pid] = uid + " " + command
                    for line in ps:
                        pid, uid, command = line.split()
                        client[pid] = uid + " " + command

                    #confronto tra le due liste
                    if not cmp(result, client) > 0: print 'PS OK'
                    else:
                        print "hidden process:"
                        key = result.keys()
                        key.sort()
                        for k in key:
                            if not client.has_key(k):
                                print k, result[k]
                            else:
                                if cmp(result[k], client[k]):
                                    print result[k], client[k]

                print
                 self._finished.wait(self._interval)
        finally:
            managerActions.closeApp(host,port)
            print 'DEBUG', self.myName + ' stopped\n'


if __name__ == '__main__':
    #avvio del thread
    c=ManagerXenVmi(sys.argv[1])
    c.run()
```
