UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA SPECIALISTICA IN TECNOLOGIE INFORMATICHE

ARCHITETTURE DI SICUREZZA E TECNOLOGIE DI VIRTUALIZZAZIONE: RILEVAMENTO DELLE INTRUSIONI TRAMITE INTROSPEZIONE

Tesi di Laurea Specialistica

Daniele Sgandurra

Relatore: Prof. Fabrizio Baiardi

Controrelatore: Prof. Antonio Cisternino

ANNO ACCADEMICO 2005-2006

UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA SPECIALISTICA IN TECNOLOGIE INFORMATICHE

ARCHITETTURE DI SICUREZZA E TECNOLOGIE DI VIRTUALIZZAZIONE: RILEVAMENTO DELLE INTRUSIONI TRAMITE INTROSPEZIONE

Tesi di

Daniele Sgandurra

Relatore:

Prof. Fabrizio Baiardi . . . . . .

Controrelatore:

Prof. Antonio Cisternino . . . . . .

Candidato:

Daniele Sgandurra . . . . . .

UNIVERSITÀ DI PISA

13 OTTOBRE 2006

Sommario

Negli ultimi tempi c'è stato un rinnovato e crescente interesse per la virtualizzazione, il cui compito è quello di creare degli ambienti di esecuzione software (chiamati anche *macchine virtuali*) tramite l'astrazione delle risorse.

Questa tecnologia fornisce degli strumenti utili a rilevare le intrusioni e gli attacchi portati ad un sistema informatico: infatti, è possibile avere una visione completa, e a più livelli, dello stato dell'*host* che viene eseguito all'interno di un ambiente virtuale. Ad esempio, la virtualizzazione permette di esaminare lo stato della memoria, dei registri del processore e dei dispositivi di I/O della macchina virtuale. Inoltre, queste informazioni sullo stato della macchina virtuale sono meno suscettibili di manomissione da parte di un attaccante.

La virtualizzazione offre anche strumenti efficaci che permettono di reagire ad attacchi e ad intrusioni: ad esempio, è possibile sospendere l'esecuzione o salvare lo stato della macchina virtuale su cui l'host è in esecuzione, per potere eseguire successivamente controlli più accurati sul suo stato.

Questa tesi presenta un'architettura per il rilevamento delle intrusioni su macchine virtuali che fa uso sia di tecniche di *introspezione*, per analizzare lo stato delle macchine virtuali tramite il *Virtual Machine Monitor*, che dei tradizionali metodi per il rilevamento delle intrusioni. L'architettura è distribuita su più macchine virtuali: una di queste ha funzionalità di introspezione e di controllo delle altre macchine virtuali e, in caso di intrusione, può agire sullo stato di esecuzione delle macchine virtuali, ad esempio bloccandone l'esecuzione.

Ringraziamenti

Innanzitutto voglio ringraziare la mia famiglia per avermi dato la possibilità di raggiungere questo importante obiettivo. Vorrei inoltre esprimere la mia gratitudine al prof. Fabrizio Baiardi per il prezioso e continuo supporto che mi ha dato durante la preparazione e la stesura della tesi. Infine, un saluto va a tutti gli amici con cui ho seguito i corsi durante questi anni: Antonio Fuschetto, Costantino Lacasella, Andrea Pasquini, Antonio Rovitti, Francesco Tamberi, Giuseppe Zichittella, Romeo Zitarosa e a tutti gli altri.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

> "How often have I said to you that when
> you have eliminated the impossible,
> whatever remains, *however improbable*,
> must be the truth?"
>
> *The Sign Of Four*, 1890
> Sir Arthur Conan Doyle

I due concetti principali che hanno guidato il lavoro descritto in questa tesi sono la *virtualizzazione* e i *sistemi per il rilevamento delle intrusioni*: questi concetti hanno fornito le basi per la progettazione e la realizzazione di *Psyco-Virt*, un primo prototipo per un'architettura distribuita per il rilevamento delle intrusioni su macchine virtuali tramite introspezione. Scopo di questa introduzione è quello di descrivere brevemente questi concetti e il lavoro svolto per questa tesi.

La virtualizzazione e le macchine virtuali

Il concetto di *virtualizzazione* è uno dei più pervasivi e utilizzati in informatica. In [74] la virtualizzazione viene definita come:

> [..] a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others.

In particolare, la virtualizzazione definisce e implementa un livello di astrazione rispetto alle risorse del computer: questo livello di astrazione media tra le risorse fisiche del computer e il software che le utilizza. Questa strategia permette di creare, all'interno della stessa macchina

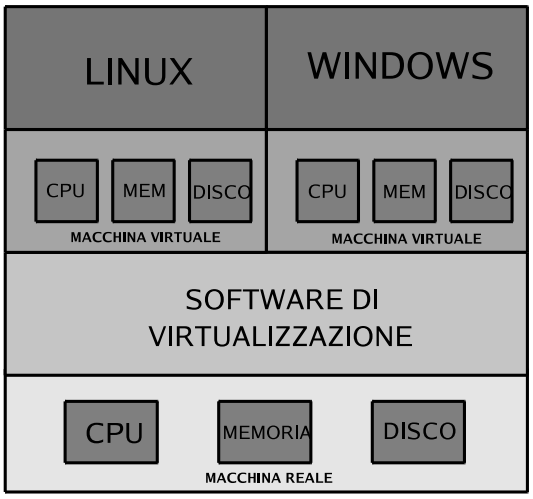


Figura 1: Esempio di virtualizzazione

fisica, più ambienti di esecuzione virtuali (le *macchine virtuali*) che emulano il comportamento della macchina reale sottostante, mettendo a disposizione dei livelli superiori l'interfaccia esportata dall'hardware della macchina.

Negli ultimi anni questa tecnologia è stata nuovamente rivalutata, soprattutto perché dà la possibilità di eseguire simultaneamente sulla stessa macchina fisica diverse istanze di sistemi operativi. Ad esempio, è possibile eseguire Linux, Windows e FreeBSD in concorrenza sulla stessa macchina, sfruttando l'esistenza di diverse macchine virtuali all'interno delle quali è possibile eseguire i sistemi operativi (vedi fig. 1).

Il software di virtualizzazione è in grado di tenere traccia dello stato di ogni sistema operativo in esecuzione nelle macchine virtuali, in maniera simile a quella con cui un sistema operativo tradizionale tiene traccia dello stato dei processi e, inoltre, fornisce *protezione* e *isolamento* tra le macchine virtuali.

Non solo si hanno benefici in termini economici, per cui, ad esempio, è possibile *consolidare* su una stessa macchina fisica diversi server, ma altresì si hanno benefici in termini di controllo centralizzato degli ambienti virtuali. È possibile salvare lo stato delle macchine virtuali o sospenderne l'esecuzione per poi ripristinarla successivamente; oppure si può *migrare* una macchina virtuale tra diverse macchine fisiche, ad esempio per facilitare il bilanciamento di carico tra diverse macchine fisiche.

Inoltre, come descritto nei par. 1.2.4 e 1.2.5, grazie alle tecnologie sviluppate recentemente da Intel e AMD, che hanno introdotto il supporto firmware per migliorare le prestazioni della virtualizzazione, è possibile eseguire in maniera più efficiente la virtualizzazione e risulta meno complesso sviluppare il software di virtualizzazione.

I sistemi per il rilevamento delle intrusioni

I sistemi per il rilevamento delle intrusioni, descritti in dettaglio nel cap. 3, hanno il compito di monitorare un sistema alla ricerca di eventi associati a tentativi di intrusione o di attacco al sistema stesso.

Tradizionalmente, i sistemi per il rilevamento delle intrusioni sono suddivisi in due categorie:

- monitoraggio di un singolo host: *Host IDS*;
- monitoraggio di un segmento di rete: *Network IDS*.

Un Host IDS è composto da uno o più *agenti* che ricercano le intrusioni analizzando le sequenze di chiamate di sistema, i file di log e monitorando le modifiche al file system: le modifiche di interesse sono quelle ai file o a agli attributi dei file, come i permessi o il proprietario. I Network IDS, invece, ricercano le intrusioni analizzando il traffico su un segmento di rete e, quindi, riescono a monitorare più host.

Inoltre, gli IDS possono essere suddivisi ulteriormente in *IDS passivi* e *IDS reattivi*. Nel caso in cui venga rilevata una possibile intrusione al sistema, un IDS passivo semplicemente registra l'evento su un file di log oppure invia un *alert* (allarme) all'interfaccia di configurazione del sistema, la *console*. In un sistema reattivo, invece, l'IDS può eseguire azioni in risposta ai tentativi di intrusione, ad esempio può aggiungere delle regole al firewall per bloccare il traffico appartenente ad una connessione utilizzata per attaccare il sistema.

Nel caso di un IDS in grado di filtrare autonomamente, ed eventualmente bloccare, il traffico e di reagire ai tentativi di intrusione, ad esempio modificando l'header di un pacchetto che vìola le specifiche del protocollo utilizzato, siamo in presenza di un IPS (*Intrusion Prevention System*, sistema per prevenire le intrusioni).

Le metodologie utilizzate dai sistemi di rilevamento delle intrusioni sono tre:

1. l'*anomaly detection*: viene stabilito qual è il comportamento normale del sistema tramite un'analisi statistica delle sue caratteristiche. Ogni comportamento che si distanzi significativamente dai valori osservati viene considerato anomalo, e quindi segnalato come tentativo di intrusione o di attacco;
2. il *misuse detection*: in questo caso è presente una base di dati contenente firme di attacchi noti. L'IDS ricerca quegli eventi che corrispondono ad almeno una di queste firme;
3. lo *specification-based detection* segnala come anomali, e quindi come tentativi di intrusione, tutti quei comportamenti che deviano dalle specifiche ricevute in ingresso che codificano l'uso corretto dei protocolli e dei programmi.

Abbiamo, quindi, che un approccio “misuse detection” è di tipo *default-allow*, per cui solo gli eventi specificati sono illegali e tutti gli altri sono legali. L’approccio “specification-based”, invece, è di tipo *default-deny* e specifica tutti e soli i comportamenti legali.

Psyco-Virt

Solitamente, i tentativi di attacco al sistema, e di conseguenza i metodi di difesa, cercano di ottenere il controllo del sistema accedendo al livello più basso possibile. Ad esempio, i *rootkit* e i programmi per rilevare i rootkit col tempo sono passati dall’essere eseguiti nello spazio utente a quello kernel, nel tentativo di avere ognuno la meglio sull’altro. Nel caso in cui sia i sistemi di difesa che i sistemi di attacco siano nel livello kernel, quello più basso in cui possono essere installati, solo il sistema che riuscirà a prevedere e prevenire per primo le mosse dell’altro potrà avere pieno controllo del sistema [76].

La virtualizzazione, che rende possibile *incapsulare* completamente lo stato di un host all’interno di una macchina virtuale, permette di introdurre un altro livello: quello del software di virtualizzazione, detto anche, nel caso della virtualizzazione a livello hardware, *hypervisor*.

Grazie a questo aspetto, la virtualizzazione rende possibile esaminare lo stato di un host “da sotto”: l’*introspezione delle macchine virtuali* [44] è quella tecnica che permette di controllare lo stato corrente di una macchina virtuale in esecuzione. Per cui si può analizzare in dettaglio l’host il cui stato è incapsulato nella macchina virtuale monitorata, alla ricerca di intrusioni. Ad esempio, è possibile ricercare determinate firme associate ad attacchi direttamente nella memoria assegnata alla macchina virtuale.

Ciò consente di ottenere i seguenti benefici:

- per un attaccante è più difficile modificare i dati che vengono analizzati tramite introspezione. Infatti, nel caso in cui il sistema di rilevamento delle intrusioni sia installato sulla stessa macchina che deve monitorare, la visione della memoria può essere stata modificata *ad hoc* dall’attaccante per nascondere alcune sezioni della memoria stessa. Invece, con la virtualizzazione, anche un attaccante che sia riuscito a compromettere il sistema in esecuzione sulla macchina virtuale, non può avere accesso all’hypervisor per modificare lo stato della macchina virtuale esaminato tramite introspezione[1];

- è possibile analizzare e confrontare i dati ottenuti tramite introspezione con quelli ottenuti dall’host monitorato, utilizzando, all’interno dell’host stesso, le normali tecniche per il rilevamento delle intrusioni. Nel caso di discrepanze tra i dati ottenuti dai due livelli,

[1] ovviamente, dobbiamo considerare l’hypervisor come facente parte del *Trusted Computing Base* (TCB) del sistema [18].

siamo in presenza di un attaccante che ha modificato alcune parti del sistema per dare una visione del sistema diversa da quella reale, ad esempio per nascondere alcuni processi in esecuzione, o file presenti in una directory di sistema;

- si possono controllare, per prevenire eventuali modifiche, gli agenti, cioè le componenti che fanno parte del sistema di rilevamento delle intrusioni che sono installate sulle macchine virtuali. Ad esempio, è possibile monitorare le pagine di memoria dei processi associati agli agenti e contenenti le istruzioni, per verificare che non vengano alterate.

Il progetto *Psyco-Virt*, descritto nel cap. 4, è stato realizzato come prototipo per un'architettura di rilevamento delle intrusioni per macchine virtuali, in grado di combinare tecniche di introspezione a metodi per il rilevamento delle intrusioni su sistemi distribuiti.

Le caratteristiche principali di questo sistema sono:

a) utilizzo della tecnica dell'introspezione per accedere allo stato delle macchine virtuali e avere una visione dei sistemi in esecuzione sulle macchine virtuali da un livello più basso rispetto quello a cui può avere accesso un attaccante;

b) uso di un'architettura distribuita per il rilevamento delle intrusioni: è presente un nodo centrale, l'*engine*, che ha il compito di coordinare tutte le componenti del sistema e di ricevere, tramite il *director* le segnalazioni inviate dalle componenti del sistema in esecuzione sulle macchine virtuali, gli *agent* e i *collector*;

c) possibiltà di eseguire delle azioni specifiche in risposta agli eventi che segnalano tentativi di intrusione e di attacco, in base al livello di gravità segnalato. È possibile modificare lo stato di esecuzione di una macchina virtuale, ad esempio bloccando la sua esecuzione per poi ripristinarla successivamente dopo aver effettuato controlli più approfonditi;

d) controllo sulle intrusioni a più livelli: come già detto, l'architettura sorveglia il sistema su più livelli in modo che, anche se l'attaccante riesce a compromettere un sottosistema, l'architettura è in grado di rilevare le differenze tra i dati forniti da un livello e quelli forniti da un altro livello;

e) resistenza agli attacchi: poiché la componente centrale del sistema di monitoraggio è installata su una macchina virtuale diversa da quella monitorata, anche in caso di compromissione di un host monitorato, il sistema di rilevamento delle intrusioni non viene compromesso.

Psyco-Virt è stato realizzato usando il linguaggio di programmazione Python [19] e adoperando, come Virtual Machine Monitor, Xen [28]; Linux è stato usato come sistema operativo in esecuzione sulle macchine virtuali.

Sono stati infine effettuati dei test per verificare l'efficacia dell'architettura proposta e per calcolare l'overhead introdotto dalla virtualizzazione e dai controlli effettuati, simulando attacchi e tentativi di intrusione per ottenere l'accesso non autorizzato ai sistemi in esecuzione sulle macchine virtuali monitorate.

Struttura della tesi

Nei prossimi capitoli vengono esaminati in maniera più approfondita i concetti presentati in questa introduzione. La struttura dei capitoli è la seguente:

Capitolo 1: La virtualizzazione

Questo capitolo analizza la virtualizzazione, i benefici e le problematiche che essa introduce.

Capitolo 2: Xen

Questo capitolo descrive Xen, un esempio di tecnologia di virtualizzazione, e ne discute gli aspetti essenziali della sua architettura.

Capitolo 3: I sistemi di rilevamento delle intrusioni

Questo capitolo fornisce una panoramica generale sullo stato dell'arte riguardante gli strumenti per il rilevamento delle intrusioni e ne discute alcuni esempi.

Capitolo 4: Psyco-Virt

Questo capitolo esamina l'architettura di Psyco-Virt, un sistema distribuito per il rilevamento delle intrusioni su ambienti virtuali tramite introspezione; inoltre, viene descritto un primo prototipo di Psyco-Virt realizzato in Python che utilizza Xen come tecnologia di virtualizzazione.

Capitolo 1

La virtualizzazione

Contenuto

La virtualizzazione non è un'idea recente: infatti, già negli anni '60 il termine *macchina virtuale* indicava un'astrazione software che forniva la stessa interfaccia della macchina reale [46, 47]. In quegli anni la virtualizzazione era confinata solo ai server specializzati, proprietari: adesso sta diventando una tecnologia comune, sempre più disponibile su vari sistemi e con vari software. Oggi questo termine viene usato anche in senso più ampio, ad esempio con la tecnologia Java Virtual Machine. In ogni caso la macchina virtuale fornisce un meccanismo di astrazione che presenta un'interfaccia semplificata verso le risorse del computer.

Facendo riferimento all'usuale visione del computer, visto come strutturato su più livelli, come illustrato nella fig. 1.1, a partire dal livello hardware/firmware per terminare con quello delle applicazioni, eseguite sopra il livello del sistema operativo, il software di virtualizzazione

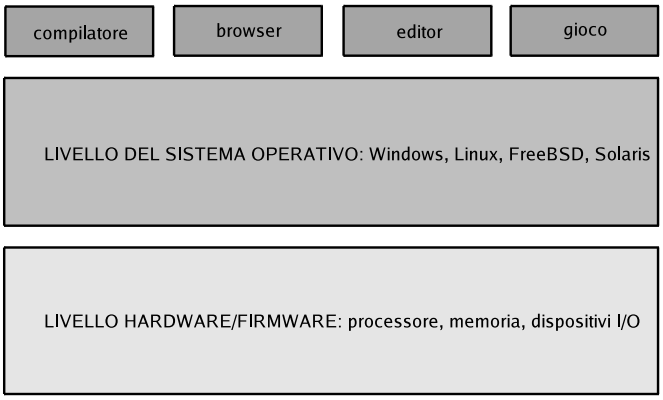


Figura 1.1: La struttura di un computer moderno

può interporsi in qualche punto tra questi livelli. In base ai livelli scelti, avremo livelli di virtualizzazione diversi. Nel seguito verranno descritti i vari livelli possibili, le strategie di implementazione e i problemi che devono essere affrontati.

1.1 Livelli di virtualizzazione

Tra i possibili livelli di virtualizzazione, tre sono meritevoli di attenzione: la virtualizzazione a livello hardware, a livello di sistema operativo e le macchine virtuali create da un linguaggio ad alto livello [69].

La virtualizzazione a livello hardware. In questo caso, il livello di virtualizzazione sta sopra il livello hardware/firmware e ha il compito di presentare al livello superiore, quello del sistema operativo, le risorse virtuali emulando le risorse fisiche che sono sulla macchina reale: il processore, la memoria e i dispositivi di I/O.

A questo livello era sviluppata la tecnologia VM/370 sui mainframe IBM (vedi la tab. 1.1).

La virtualizzazione a livello di sistema operativo. In questo caso, il software di virtualizzazione si interpone tra il sistema operativo e le applicazioni che esso esegue. Nelle macchine virtuali create è così possibile eseguire programmi scritti per lo specifico sistema operativo. Un esempio di questa tecnologia sono le Jail di FreeBSD [58].

Le macchine virtuali a livello applicativo. In quest'ultimo caso, invece, la macchina virtuale è un'applicazione scritta per un particolare sistema operativo ed esegue programmi che sono stati

La tecnologia VM/370. La tecnologia VM/370 gestisce le risorse del computer in maniera tale che ogni utente, sia in locale che in remoto, ottenga una replica virtuale di un sistema IBM System/370, compresi i dispositivi di I/O. È possibile eseguire diversi sistemi operativi in concorrenza su macchine virtuali diverse [66, 72].
Il VM/370 è formato da due componenti principali: il *Control Program* (CP) e il *Conversational Monitor System* (CMS). Il CP gestisce tutte le risorse del sistema e crea le macchine virtuali su cui possono essere eseguiti i sistemi operativi.
Una macchina virtuale fornisce risultati che sono funzionalmente equivalenti a quelli di una macchina reale. Inoltre, il CP è trasparente alle macchine virtuali, eccetto che nel caso dell'*handshaking* che permette ai sistemi operativi di avere una interfaccia diretta di comunicazione con il CP. Il codice eseguito dalle macchine virtuali non viene modificato, per cui è come se fosse eseguito su un sistema reale.
Il CMS fornisce funzionalità di supporto, ad esempio fornisce un ambiente per lo sviluppo di programmi, assieme a strumenti di *debugging* ed *editing* tra gli altri.
Tra i vari benefici che offre questa tecnologia segnaliamo:

- possibilità di eseguire più sistemi operativi in concorrenza. Ad esempio, durante una fase di aggiornamento di un sistema operativo, è possibile mantenere le due versioni, la nuova e la vecchia, attive su due macchine virtuali differenti;
- isolamento degli spazi di indirizzamento: grazie alla segmentazione e alle tabelle delle pagine, anche in caso di errori, macchine virtuali diverse non interferiscono tra di loro.

Le normali istruzioni vengono eseguite direttamente dalla macchina reale; invece, le istruzioni privilegiate, ad esempio, quelle per le operazioni di I/O, vengono simulate dal CP. Ad esempio, dopo aver schedulato una richiesta di I/O, il CP restituisce il controllo al sistema operativo della macchina virtuale che ha eseguito la richiesta.

Tabella 1.1: La tecnologia VM/370

compilati specificatamente per quella macchina virtuale, che definisce un proprio linguaggio. Come già citato, un esempio di questa tipologia di macchina virtuale è la Java Virtual Machine

1.2 La virtualizzazione a livello hardware

L'idea alla base di questa tecnologia è quella di astrarre ogni unità presente su una macchina fisica, quali il processore, la memoria e i dispositivi di I/O, creando degli ambienti di esecuzione software che emulino il comportamento della macchina sottostante su cui vengono eseguiti. Il software eseguito dalle macchine virtuali si comporta come se fosse eseguito direttamente sulla macchina reale.

Questo tipo di astrazione è concettualmente simile a quella fornita dalla memoria virtuale:

infatti, la memoria virtuale dà ai processi l'illusione di possedere uno spazio di memoria contiguo e unidirezionale a partire dall'indirizzo 0 e, in questa maniera, semplifica la visione che i processi hanno della memoria e nasconde loro le locazioni della memoria fisica, che non occorre che siano contigue.

Similmente, una macchina virtuale fornisce ai sistemi operativi l'astrazione della macchina fisica su cui la macchina virtuale stessa è eseguita. In questo modo, l'astrazione fornita dalla macchina virtuale permette ai sistemi operativi di comportarsi come se fossero eseguiti direttamente sulla macchina reale.

La ragione principale per adottare questa strategia è quella di far condividere il medesimo hardware ad un numero elevato di sistemi operativi, che possono essere eseguiti concorrentemente su diversi ambienti d'esecuzione sulla stessa macchina fisica.

Il livello software che si occupa della creazione delle macchine virtuali è detto *Virtual Machine Monitor* (VMM) o anche *Hypervisor*, ed è un piccolo strato di software che viene eseguito direttamente sopra l'hardware della macchina. Una installazione di questo tipo è detta *bare metal*.

1.2.1 Virtual Machine Monitor

Il VMM crea l'illusione dell'esistenza di molte macchine virtuali su una singola macchina reale. Poiché viene esportata la stessa interfaccia fornita dal livello hardware sottostante, ogni macchina virtuale può eseguire tutti i programmi scritti per quella macchina fisica. Di conseguenza è possibile eseguire più sistemi operativi simultanenamente sulla stessa macchina fisica, grazie al fatto che ogni sistema operativo è eseguito da una macchina virtuale, come illustrato nella fig. 1.2.

Il software eseguito da una macchina virtuale si comporta come se fosse eseguito direttamente dall'architettura fisica e come se avesse pieno controllo di essa. In questo modo, il VMM è trasparente al software eseguito dalle macchine virtuali.

Il VMM deve garantire le seguenti proprietà:

- *compatibilità*: tutte le applicazioni scritte per la macchina reale, virtualizzata dal VMM, devono poter essere eseguite anche nelle macchine virtuali;
- *isolamento*: ogni macchina virtuale deve essere isolata da ogni altra, quindi sia da attacchi accidentali che malintenzionati. Non deve essere possibile che una macchina virtuale interferisca con un'altra macchina e che siano violati i meccanismi di protezione implementati dal VMM.

Inoltre, grazie alla proprietà dell'*incapsulamento*, che garantisce che tutto il software

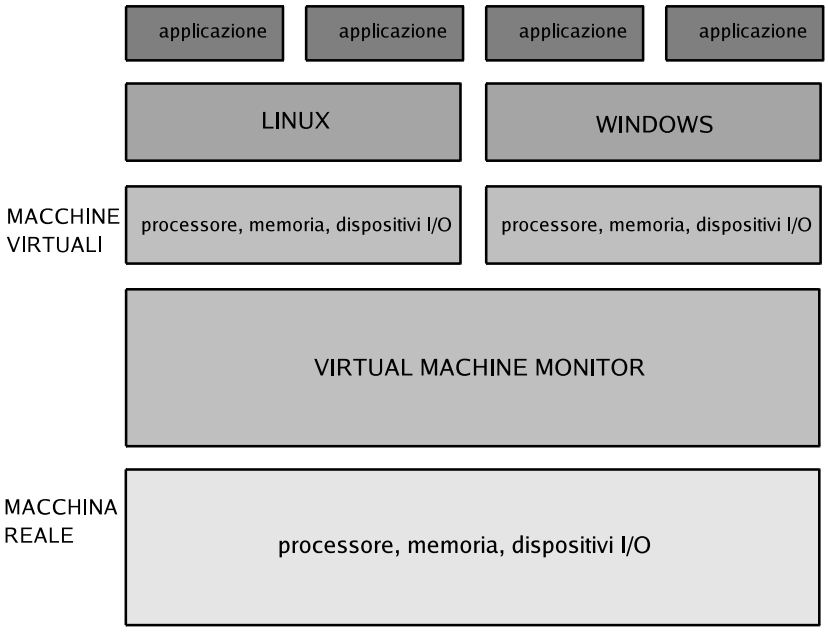


Figura 1.2: Il virtual machine monitor

venga eseguito all'interno delle macchine virtuali, solo il VMM ha accesso alle risorse fisiche, permettendo così di separare in maniera netta le macchine virtuali dall'hardware. Grazie a questa proprietà è possibile eseguire più macchine virtuali su una stessa macchina fisica. È inoltre possible sospendere l'esecuzione di una macchina virtuale o migrare una macchina virtuale da una macchina reale ad un'altra. Le risorse virtuali vengono così mappate dinamicamente sulle risorse fisiche, in base alle necessità e ai parametri da ottimizzare.

La *performance* effettiva di un sistema incapsulato in una macchina virtuale è molto vicina a quella nativa, cioè senza il livello di virtualizzazione. Infatti, le due interfacce, quella virtuale e quella reale, sono identiche o molto simili, ed è quindi facile mappare le richieste dall'una all'altra. Inoltre, il VMM entra in gioco solamente quando deve prendere controllo della macchina virtuale, ad esempio per mantenere l'isolamento. Negli altri casi, la macchina virtuale utilizza tutto il suo tempo usando le risorse in maniera diretta, poiché le istruzioni non privilegiate vengono eseguite direttamente sulla macchina reale (vedi dopo).

Inoltre, grazie alla dimensione ridotta di un VMM rispetto ad un sistema operativo moderno, in termini di numero di istruzioni, il VMM è più efficiente nell'esecuzione di alcune operazioni, ad esempio la protezione, ed è anche più facile verificarne formalmente la correttezza.

1.2.2 Full virtualization e paravirtualization

Esiste anche una distinzione tra le tipologie di virtualizzazione che vengono fornite. Abbiamo virtualizzazione completa (*full virtualization*) quando un sistema operativo può essere eseguito nella macchina virtuale creata dall'hypervisor senza che sia necessario modificare il codice. Invece, nel caso di paravirtualizzazione (*paravirtualization*) sono necessarie alcune modifiche al codice del sistema operativo per "portarlo", cioè per permettergli di essere eseguito da una macchina virtuale.

In quest'ultimo caso, il VMM presenta un'interfaccia di astrazione, tramite la macchina virtuale, non uguale ma simile all'interfaccia esportata dall'hardware sottostante. Sono quindi necessarie delle modifiche al codice del sistema operativo, ma non alle applicazioni che vengono eseguite sul sistema operativo.

Uno dei vantaggi della paravirtualizzazione è una performance migliore. Infatti, non è necessario effettuare la traduzione dinamica delle istruzioni. Uno svantaggio, invece, è che il numero di sistemi operativi che possono essere eseguiti con questa tecnologia è minore, in quanto possono essere eseguiti nelle macchine virtuali solo quei sistemi operativi dei quali è possibile modificare il codice.

1.2.2.1 Binary translation e dynamic recompilation

Tra le varie tecniche utilizzate per implementare la virtualizzazione completa citiamo quella della *binary translation*, tra gli altri usata da VMware [26] e da Virtual PC della Microsoft [14]. Questa tecnica prevede una traduzione a tempo di esecuzione del codice destinato all'architettura hardware. In pratica, visto che il codice del sistema operativo non viene modificato, tutte le parti di codice che contengono operazioni privilegiate, e che quindi richiedono l'intervento del VMM, devono essere interpretate a tempo di esecuzione per inserirvi *trap* al VM. Questa soluzione, rispetto al quella della paravirtualizzazione, ha però un overhead più elevato.

Un'altra tecnica per implementare la virtualizzazione completa è la *dynamic recompilation*. Questo termine viene usato per indicare il fatto che un sistema operativo, progettato per essere eseguito su una particolare classe di CPU e quindi con un determinato linguaggio macchina, viene eseguito da una macchina virtuale che emula un altro tipo di CPU, con un differente insieme di istruzioni. Di conseguenza, a tempo di esecuzione parti di codice vengono ricompilate per essere eseguite sulla nuova architettura. Spesso questa compilazione permette di applicare altre ottimizzazioni.

Nella "binary translation" esiste il classico ciclo di prelievo-decodifica-esecuzione dell'istruzione, e quindi ogni istruzione viene emulata per la nuova architettura. Nel caso della "dynamic recompilation", invece, durante l'esecuzione vengono riscritte in blocco sezioni di codice, ad

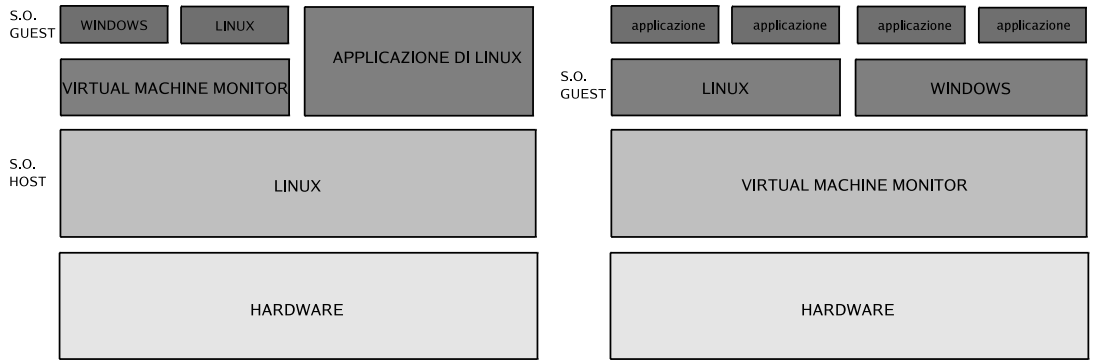


Figura 1.3: Hosted VMM (a); Unhosted VMM (b)

esempio la prima volta che si entra in una determinata sezione di codice. Successivamente, se quelle istruzioni verrano rieseguite, si utilizzerà nuovamente la sezione ricompilata. Un'altra differenza è dovuta al fatto che, nel caso della "binary translation" le istruzioni che vengono eseguite direttamente dalle macchine virtuali non vengono tradotte; invece, nel caso della "dynamic recompilation" tutte le istruzioni devono essere emulate.

1.2.3 Problemi posti dall'architettura hardware

Alcune istruzioni del linguaggio macchina dell'architettura IA32, considerata in questa tesi, aumentano la complessità della virtualizzazione completa [68]. Ad esempio il processore, nel caso in cui venga eseguita l'istruzione `POPF` in modalità non privilegiata, non rimuove la parola dalla testa dello stack per salvarla nel registro `EFLAGS`, come farebbe in modalità privilegiata, e inoltre non genera una trap. Di conseguenza, se un sistema operativo eseguisse questa istruzione in modalità non privilegiata, non otterrebbe l'effetto desiderato e inoltre il VMM non sarebbe notificato del tentativo del sistema operativo di eseguire quella istruzione [70].

Inoltre, l'architettura IA32 prevede che il *Translation Lookaside Buffer* (TLB) venga gestito direttamente a firmware, per cui, per ottenere una virtualizzazione completa, è necessario che il VMM sia eseguito come applicazione su un sistema operativo che è in esecuzione direttamente sull'hardware. In questo caso si parla di *Hosted VMM* (vedi fig. 1.3(a)). Altrimenti, si deve utilizzare la tecnica della paravirtualizzazione e si parla di *Unhosted VMM* (vedi fig. 1.3(b)). Il sistema operativo che esegue il VMM viene detto *Host Operating System* (sistema operativo ospitante) e i sistemi operativi che vengono eseguiti sul VMM, sono detti *Guest Operating System* (sistemi operativi residenti).

Ultimamente però, grazie a tecnologie quali *Intel Virtualization Technology* [10] e *Amd*

Pacifica Virtualization Technology [1], la complessità di virtualizzazione dovuta all'architettura hardware/firmware viene significativamente ridotta (vedi par. 1.2.4 e par. 1.2.5).

1.2.3.1 I livelli di privilegio

I processori dell'architettura IA32 supportano politiche di protezione che utilizzano il concetto di *ring*. Ad ogni ring è associato un *livello di privilegio*, un valore a 2-bit. Il software eseguito al livello 0 gode dei privilegi più alti, mentre quello con il livello 3 di quelli più bassi. Il livello di privilegio determina, ad esempio, se le istruzioni privilegiate possono essere eseguite senza errori.

Solitamente, per potere avere pieno controllo del processore, il sistema operativo deve poter essere eseguito al livello di privilegio 0 (vedi fig. 1.4(a)). Ma, poiché un VMM non può concedere tale controllo ad un sistema operativo, i sistemi operativi "guest" non possono essere eseguiti a questo livello. In questi casi, viene usata le tecnica del *ring deprivileging* [77], che permette di eseguire un sistema operativo "guest" ad un livello più alto del livello 0.

Ci sono due modelli con questa tecnica:

1. il modello 0/1/3 (vedi fig. 1.4(b)), in cui i sistemi operativi "guest" vengono eseguiti ai livelli 1 e 3, il livello 1 per il sistema operativo, il 3 per le applicazioni;

2. il modello 0/3/3 (vedi fig. 1.4(c)), in cui sia i sistemi operativi "guest" che le applicazioni vengono eseguite al livello 3.

Questa tecnica però determina alcuni problemi nella realizzazione della virtualizzazione [77], per cui negli ultimi tempi i produttori di processori hanno sviluppato tecnologie che forniscono un supporto hardware/firmware per la virtualizzazione, semplificando la realizzazione dei VMM. Ad esempio, con la Intel Virtualization Technology e la Amd Pacifica Virtualization Technology non è più necessario effettuare la traduzione delle istruzioni o modificare il codice del sistema operativo, rendendo di fatto possibile l'esecuzione di un più ampio numero di sistemi operativi sul VMM, con alti livelli di performance grazie all'aggiunta di nuove istruzioni al set d'istruzioni del processore.

1.2.4 Intel Virtualization Technology

Scopo centrale della *Virtualization Technonology* (VT) della Intel, è quello di far sì che non sia più necessario, per i sistemi di virtualizzazione, di adottare tecniche come la "paravirtualization" o la "binary translation". Inoltre, questa tecnologia semplifica lo sviluppo dei VMM e permette di supportare un numero maggiore di sistemi operativi.

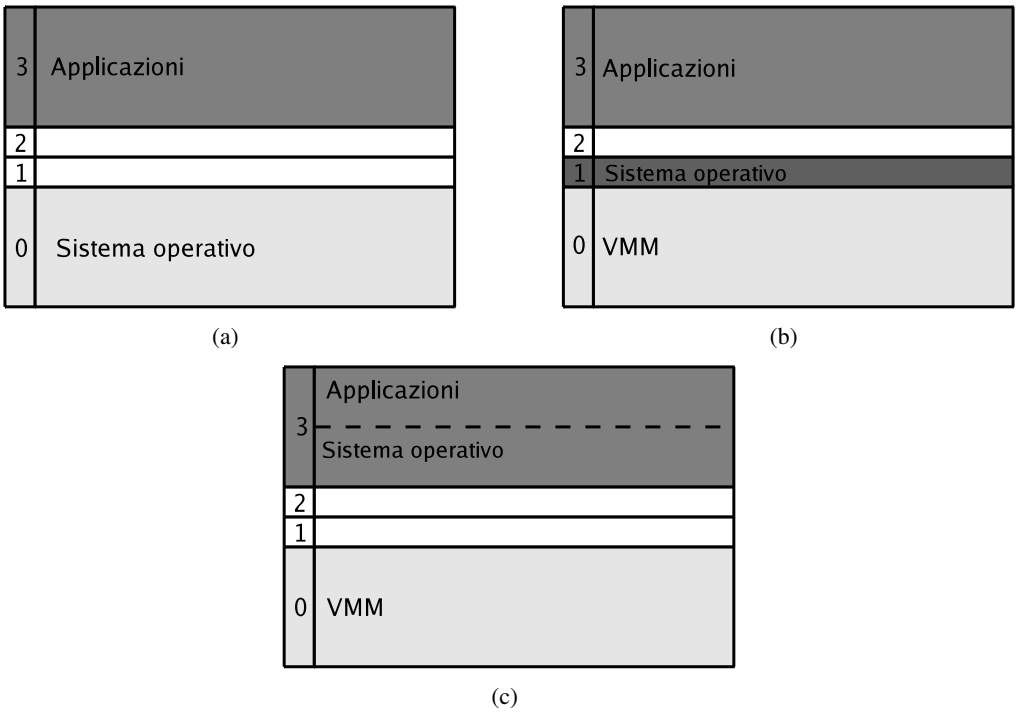


Figura 1.4: Sistema non virtualizzato (a); sistema virtualizzato con modello 0/1/3 (b); sistema virtualizzato con modello 0/3/3 (c).

Due sono le tecnologie Intel collegate alla virtualizzazione: la tecnologia *VT-x*, che fa riferimento alle estensioni per la virtualizzazione sui processori IA-32, e quella *VT-i* che, invece, fa riferimento alle estensioni sulla famiglia dei processori Itanium. Il paragrafo successivo descrive brevemente l'architettura VT-X [62].

1.2.4.1 L'architettura VT-x

Questa architettura introduce due nuove modalità di esecuzione per il processore: la *VMX root operation* e la *VMX non-root operation*. La prima permette di riprodurre, in maniera simile, il comportamento dei processori IA-32 senza VT-x, ed è una modalità di funzionamento pensata ad uso del VMM. La seconda modalità di operazione, invece, fornisce un ambiente di esecuzione controllato da un VMM ed è pensata per facilitare la creazione delle macchine virtuali. In entrambe le modalità sono presenti i quattro livelli di privilegio.

Oltre a questo, la tecnologia VT-x introduce due nuove transizioni:

- *VM ENTRY*: transizione da un'operazione "VMX root" ad una "VMX non-root";
- *VM EXIT*: transizione da un'operazione "VMX non-root" ad una "VMX root".

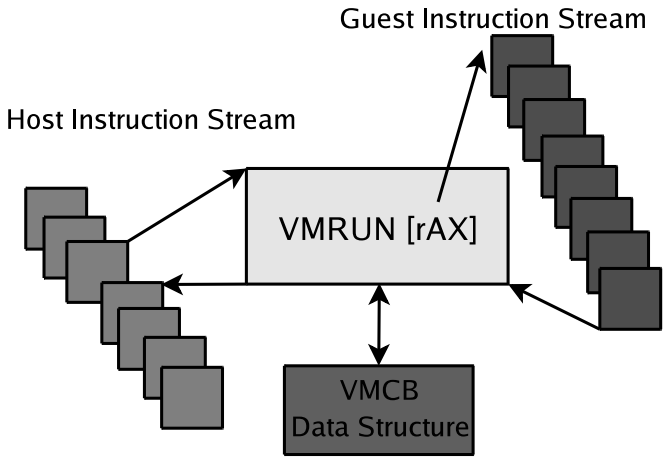


Figura 1.5: L'istruzione VMRUN

Queste due transizioni sono possibili grazie anche ad una nuova struttura dati chiamata *Virtual Machine Control Structure* (VMCS), una struttura contenenti due aree, la *guest-state area* e la *host-state area*. Queste aree contengono dei campi che permettono di salvare lo stato del processore. La transizione "VM Entry" carica lo stato del processore dall'area guest-state. La transizione "VM Exit", invece, salva lo stato del processore nell'area guest-state e carica lo stato del processore dall'area host-state.

Alcune istruzioni e alcuni eventi causano una "VM Exit". Alcune istruzioni possono essere configurate per provocare l'uscita, altre, se eseguite, la causano sempre.

1.2.5 Amd Pacifica Virtualization Technology

In questa architettura [1, 79], l'istruzione *VMRUN* permette di passare dalla modalità in cui viene eseguito il flusso di istruzioni dell'"host" a quella in cui vengono eseguite le istruzioni del "guest" e, una volta che il "guest" ha terminato, si ritorna all'istruzione dell'"host" seguente la VMRUN (vedi fig. 1.5).

Per fare questo, viene anche utilizzata una nuova struttura dati, la *Virtual Machine Control Block* (VMCB). La struttura contiene parametri che determinano quali azioni fanno sì che si esca dalla modalità "guest" e si torni a quella "host". Tale transizione può essere provocata anche esplicitamente eseguendo l'istruzione *VMMCALL*. Tutte le informazioni di stato del processore di un "guest" vengono salvate nel VMCB.

Quando viene eseguita l'istruzione VMRUN:

- lo stato dell'"host" viene salvato in memoria;
- lo stato del "guest" viene caricato dal VMCB;

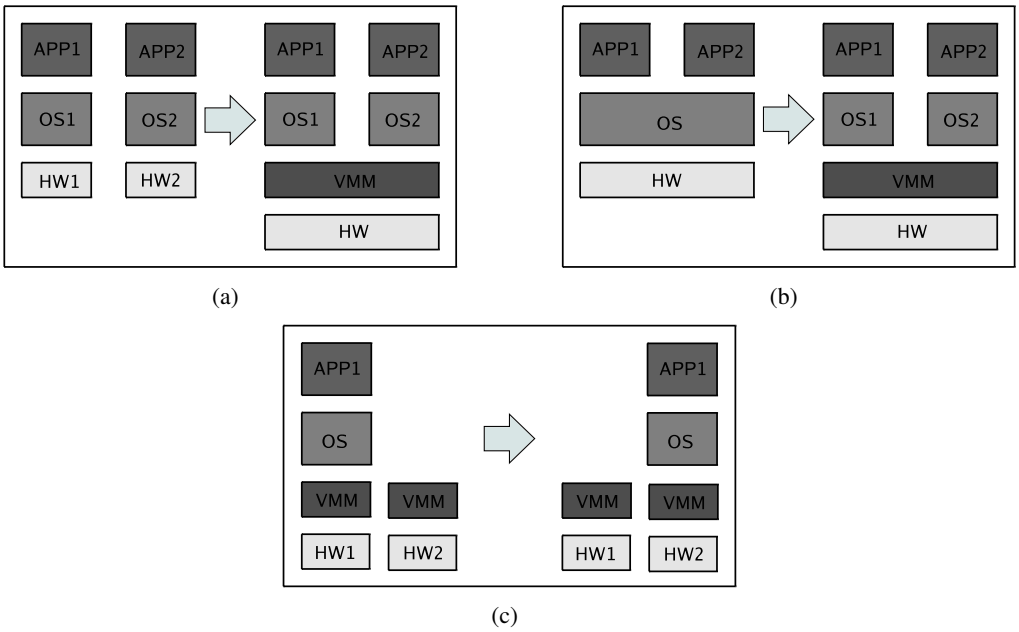


Figura 1.6: Consolidamento (a); isolamento (b); migrazione (c).

- viene messo in esecuzione il "guest".

All'uscita dall'istruzione VMRUN:

- lo stato del "guest" viene salvato nel VMCB;
- lo stato dell'"host" viene caricato dalla memoria.

Viene utilizzato anche un *MSR* (Model Specific Register), il `vm_hsave_pa`, per permettere di salvare lo stato dell'"host".

1.3 Benefici

Alcuni dei vantaggi derivanti dalla virtualizzazione sono elencati di seguito [77, 69]:

- **consolidamento** (fig. 1.6(a)): si possono installare più sistemi operativi su una singola macchina, facilitandone la gestione (tramite controllo centralizzato);
- **risparmio**: grazie al consolidamento, è possibile avere poche macchine reali su cui vengono eseguite molteplici macchine virtuali;
- **ottimizzazione delle risorse**: invece di avere tante macchine reali sottoutilizzate (ad esempio, su ognuna delle quali sono in esecuzione solo alcune applicazioni), è possibile creare tante macchine virtuali per ogni macchina reale sottoutilizzata;

- **isolamento** (fig. 1.6(b)): è possibile garantire un alto livello di protezione tra le varie macchine virtuali; ad esempio, si possono separare dati con diversi livelli di confidenzialità sopra un unico sistema tramite le macchine virtuali [16];
- **affidabilità** e **sicurezza**: i "crash" delle applicazioni su una macchina virtuale non interferiscono con le altre macchine virtuali. Inoltre, le intrusioni possono essere confinate solo nella singola macchina virtuale attaccata;
- **facilità di creazione di macchine virtuali**: ad esempio, una macchina virtuale può essere creata semplicemente copiando un file contenente l'immagine del sistema operativo e il file system;
- **migrazione** (fig. 1.6(c)): poiché lo stato di un sistema operativo (guest) è incapsulato totalmente in una macchina virtuale, è possibile far sì che il sistema migri da una macchina reale ad un'altra, ad esempio per migliorare il bilanciamento del carico;
- è possibile **sospendere** e **ripristinare** lo stato di un sistema operativo per tornare ad un punto precedente della sua storia, operazione detta di *rollback*;
- è possibile creare delle macchine virtuali per il **testing** di nuovi sistemi operativi, o **sandbox** per applicazioni.

1.4 La sicurezza negli ambienti di virtualizzazione

Tra i benefici offerti dalla virtualizzazione, uno è dato dalla possibilità di definire a livello di VMM delle politiche di controllo degli accessi sia per isolare le macchine virtuali che per condividere le risorse: compito dell'hypervisor è quello di far rispettare queste politiche. La virtualizzazione introduce però anche nuovi problemi che non sono presenti negli ambienti reali. Nei seguenti paragrafi verranno analizzati i benefici e i problemi relativi alla sicurezza introdotti con la virtualizzazione.

1.4.1 Politiche MAC

Uno degli aspetti più interessanti della virtualizzazione, è la possibilità di implementare al livello del VMM dei meccanismi di sicurezza per controllare il flusso di informazioni tra le varie macchine virtuali; è possibile cioè controllare la condivisione delle risorse tra le varie macchine virtuali.

Al giorno d'oggi, molti servizi quali i Web Services, le Intranet, i Grid Computing, richiedono una infrastruttura che sia inerentemente sicura: ma i componenti presenti in tale infrastruttura,

ad esempio i web server, i database, i browser, hanno spesso differenti requisiti di sicurezza che possono, talvolta, essere in conflitto tra di loro. Per questo, tali componenti devono essere isolati tra di loro, e ad ognuno di essi devono essere applicati differenti livelli di controllo.

Poiché, solitamente, per applicare questi controlli ci si affida agli strumenti messi a disposizione dai sistemi operativi, si hanno due possibilità:

1. le applicazioni che hanno differenti requisiti di sicurezza vengono eseguite su sistemi fisici diversi;

2. ci si affida ai meccanismi di sicurezza messi a disposizione dal sistema operativo.

Putroppo, i controlli disponibili sui sistemi operativi moderni non riescono a garantire appieno l'isolamento tra le varie applicazioni. Ciò è dovuto a vari motivi, tra cui:

- spesso i controlli sono di tipo *DAC* (Discretionary Access Control): è compito dell'utente assegnare i diritti alle risorse, per cui non viene risolto il problema di separare ciò che l'utente esegue intenzionalmente da ciò che l'utente esegue non intenzionalmente, ad esempio i virus;

- la dimensione, in numero di istruzioni, degli attuali sistemi operativi, ha raggiunto valori estremamente elevati ed è quindi difficile verificare formalmente la correttezza dei meccanismi di protezione;

- le applicazioni affette da vulnerabilità, se autorizzate ad eseguire compiti critici, ad esempio quelli che richiedono diritti di amministratore, possono compromettere la sicurezza del sistema.

Recentemente, SELinux [21] ha introdotto la possibilità di definire e implementare le politiche *MAC* (Mandatory Access Control) nel sistema operativo Linux. Anche in questo caso, però, rimane il problema che il sistema operativo è sempre passibile di attacchi che permettono all'attaccante di ottenere i privilegi di amministratore e, quindi, possono mettere a rischio la sicurezza complessiva del sistema.

Per mitigare questi problemi, è possibile delegare i controlli di accesso al VMM, come illustrato in fig. 1.7, che può implementare le politiche MAC. Queste politiche di sicurezza sono definite e controllate a livello di sistema, invece che a livello di utente, e hanno i seguenti obiettivi:

- isolare le macchine virtuali, per quanto riguarda gli eventuali attacchi;

- condividere le risorse tra le macchine virtuali in modo sicuro.

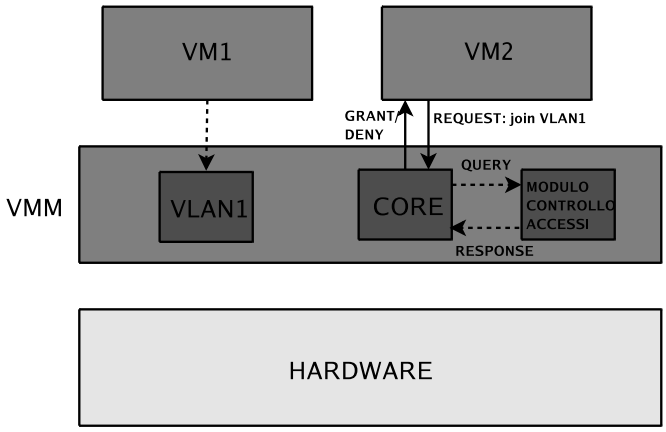


Figura 1.7: Esempio del modulo di controllo accessi

Ad esempio, è possibile definire delle classi di macchine virtuali, ognuna con differenti livelli di sicurezza, e descrivere in maniera formale quali sono i requisiti di sicurezza per poter condividere le risorse tra e all'interno di tali classi. In questi casi, il VMM deve essere in grado di:

- interpretare la descrizione formale dei requisiti di sicurezza;
- applicare i controlli di protezione.

Si rimanda al cap. 2.6 per una descrizione più dettagliata dell'implementazione di questo metodo su Xen.

In questa maniera, i sistemi operativi in esecuzione su una stessa macchina possono cooperare tra di loro per offrire un servizio comune, ad esempio i servizi distribuiti prima elencati, in maniera controllata. Un sistema operativo compromesso non interferirà con gli altri sistemi operativi residenti sulla stessa macchina fisica. Inoltre, il numero ridotto di istruzioni del modulo di controllo accessi rende più facile il controllo di verifica formale di correttezza del codice. Si consideri, ad esempio, che il modulo di controllo accessi di Xen comprende circa 3000 istruzioni.

Il passo successivo nell'adozione di questa tecnologia per i problemi di sicurezza, è quello di permettere di controllare la condivisione delle risorse applicando le stesse politiche MAC tra differenti macchine fisiche [43, 64]. Cioè, realizzare un meccanismo di controllo distribuito (*Distributed Reference Monitor*), che permetta di descrivere e applicare le stesse politiche di controllo degli accessi tra macchine virtuali allocate su più macchine fisiche connesse via rete.

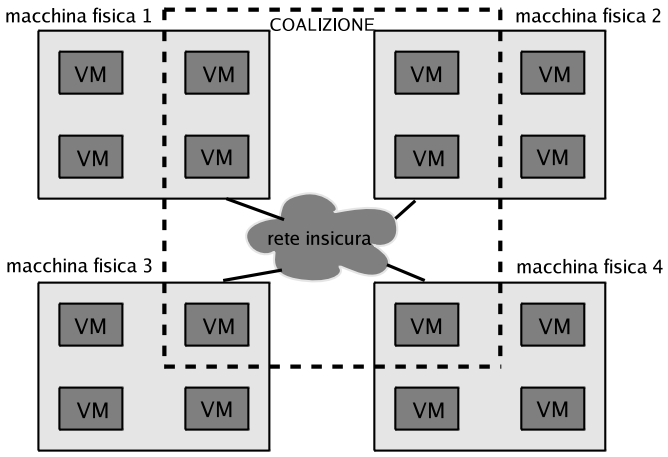


Figura 1.8: Una coalizione di macchine virtuali

1.4.1.1 Reference Monitor distribuito

La figura 1.8 illustra l'implementazione di questa tecnologia realizzata dall'IBM con il sistema DeuTeRiuM [64]: siamo in presenza di un'applicazione distribuita che è formata da quella che viene detta una *coalizione*, un insieme di macchine virtuali con la stessa politica di controllo accessi. Le macchine virtuali possono essere distribuite su diverse macchine fisiche.

Ogni macchina fisica dispone del suo *reference monitor* [30] che applica le politiche MAC tra le sue macchine virtuali. Un reference monitor, deve soddisfare i seguenti requisti:

a) deve essere *tamperproof*, cioè deve essere in grado di proteggersi da manomissioni;

b) deve fornire *mediazione completa* per tutte le operazioni critiche per la sicurezza del sistema;

c) deve essere di *dimensioni ridotte* in modo che possa esserne data una validazione formale della sua correttezza.

L'insieme dei reference monitor, dà vita ad un meccanismo distribuito in grado di proteggere una coalizione di macchine virtuali da altre coalizioni, e di proteggere le macchine virtuali all'interno della stessa coalizione da attacchi interni alla coalizione. Deve essere possibile anche aggiungere nuove macchine virtuali alla coalizione e migrarle tra una coalizione e un'altra.

La figura 1.9 illustra l'architettura generale di questo sistema: ogni macchina virtuale ha un'etichetta che identifica la coalizione cui la macchina virtuale appartiene. Le macchine virtuali con la stessa etichetta formano una coalizione e possono cooperare tra di loro e condividere la stessa politica MAC, cioè la stessa etichetta MAC.

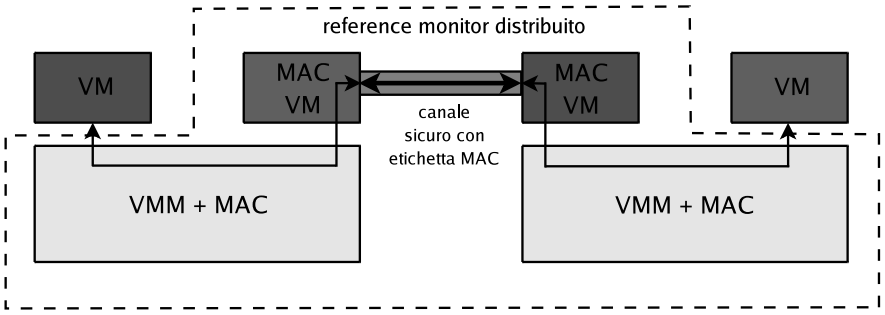


Figura 1.9: Esempio di reference monitor distribuito

Alcune macchine virtuali sono speciali, nel senso che possono appartentere a più coalizioni. Ad esempio, la MAC VM può far parte di tutte le coalizioni: ciò è dovuto al fatto che essa è in possesso di un'etichetta particolare per indicare questa proprietà.

Il reference monitor distribuito è l'unione di tutti i reference monitor in esecuzione sulle macchine fisiche dove risiedono le macchine virtuali che fanno parte della coalizione. L'unione delle politiche MAC applicate dai singoli reference monitor forma la politica MAC comune all'interno della coalizione. Il reference monitor distribuito deve applicare tutte le politiche MAC della coalizione nei confronti delle altre coalizioni e, visto che le coalizioni possono contenere macchine virtuali presenti sulle stesse macchine fisiche, il reference monitor distribuito deve garantire anche proprietà quali l'isolamento tra macchine virtuali appartenenti a coalizioni diverse ma eseguite sulla stessa macchina fisica.

Le comunicazioni tra le varie macchine fisiche avvengono su un canale sicuro, anch'esso in possesso di una sua etichetta MAC. È compito del reference monitor distribuito costruire tutti i canali di comunicazione: questi canali proteggono la segretezza delle comunicazioni e garantiscono l'integrità dei dati che passano su di essi. Poiché il canale ha una sua etichetta MAC, entrambe le macchine fisiche, su cui risiedono i punti terminali del canale, riescono a stabilire quali sono le macchine virtuali che possono usare quel canale.

È possibile aggiungere altre macchine virtuali al reference monitor distribuito, tramite un'operazione chiamata *bridging* che è composta da quattro fasi:

1. viene stabilita una politica MAC comune: il reference monitor che sta per entrare a far parte della coalizione, cioè quello che appartiene alla macchina fisica che esegue la macchina virtuale che vuole unirsi alla coalizione, ottiene la politica MAC della coalizione, controlla che essa sia compatibile con le sue politiche e, successivamente, aggiunge la propria politica a quella della coalizione. La politica risultante sarà la nuova politica MAC della coalizione;

2. vengono usate tecniche di *attestazione remota* per verificare, in maniera mutuamente affidabile, la proprietà dei reference monitor di essere "tamperproof" e di essere in grado di fornire la mediazione completa. Queste proprietà devono essere verificate sia da parte del reference monitor entrante nella coalizione che da parte del reference monitor distribuito. Ciò richiede che siano fornite attestazioni sul codice del modulo di controllo degli accessi e sulle politiche MAC implementate dai singoli reference monitor;

3. vengono inizializzate le macchine virtuali e vengono assegnate loro differenti etichette MAC basate sull'attestazione del codice;

4. infine, viene costruito il canale sicuro tra le macchine fisiche: al momento di effettuare la comunicazione, viene selezionata anche la politica di sicurezza riguardante la comunicazione, che definisce, ad esempio, il tipo di cifratura usata e quale algoritmo di hash, e anche l'etichetta MAC per il canale.

1.4.2 Problematiche di sicurezza nelle macchine virtuali

La facilità con cui è possibile creare macchine virtuali, è ovviamente un vantaggio per gli utenti, ma può far sì che il numero di macchine virtuali all'interno di un ambiente, ad esempio all'interno di un'organizzazione, cresca in maniera incontrollata. Ciò può avere ripercussioni negative sulla sicurezza. Alcuni esempi di questi aspetti negativi sono [45]:

- in caso di eventi catastrofici, ad esempio attacchi dovuti a worm, si possono avere degli impatti devastanti se tutte le macchine virtuali vengono colpite, moltiplicando quindi la portata degli effetti dell'attacco. In questo scenario, il numero di macchine virtuali da controllare alla ricerca delle vulnerabilità, ed eventualmente su cui eseguire operazioni di installazione di patch per la rimozione del codice maligno, può essere così elevato da rendere questo compito estremamente oneroso;

- possono sorgere particolari fenomeni che fanno sì che un numero elevato di macchine virtuali appaia e scompaia dalla rete in maniera discontinua; se una di questa macchine è infetta da un worm e, mentre è attiva, ne infetta altre e scompare prima di essere identificata, il compito di identificare tutte le macchine colpite dal worm è particolarmente difficile;

- la possibilità di creare dei "checkpoint" (punti di ripristino) e di eseguire poi dei "rollback" per tornare a configurazioni precedenti di una macchina virtuale, fa sì che lo stato di un sistema operativo non abbia più una storia lineare. Senza la virtualizzazione, se vengono installate tutte le patch disponibili a ritmi regolari il sistema operativo è in ogni momento

Nome	Metodo di operazione	Sistem operativi "host"	Sistemi operativi "guest"
Denali [80]	paravirtualizzazione	NetBSD, Linux	NetBSD, Ilwaco
Jail [58]	virtualizzazione a livello di S.O.	FreeBSD	FreeBSD
Microsoft Virtual PC 2004 [14]	binary translation	Windows 2000/XP Professional	MS Dos, Windows 9x/Me/2000/XP/NT, OS/2
QEMU [20]	dynamic recompilation	Windows, Linux, OS/X, FreeBSD, BeOS	Windows, Linux, OS/X, xBSD
UML (User Mode Linux) [24]	modifica al kernel di Linux	Linux	Linux
Virtuozzo [25]	virtualizzazione a livello di S.O.	OS/X	Windows, OS/2, Linux
VMware ESX Server 3.0 [26]	binary translation	Bare Metal	Windows, Linux, FreeBSD, Netware
VMware Workstation 5.5 [27]	binary translation	Windows, Linux	Dos, Windows, Linux, FreeBSD, Netware, Solaris
Xen [28]	paravirtualizzazione	Linux, NetBSD	Linux, xBSD, Windows XP/2003 (con processori IVT o Pacifica)

Tabella 1.2: Software di virtualizzazione

più aggiornato e "sicuro" rispetto al periodo precedente; invece, in ogni momento lo stato di una macchina virtuale può essere descritto tramite un albero, con differenti punti di scelta che possono essere eseguiti contemporaneamente causando delle biforcazioni. Di conseguenza, la possibilità di tornare indietro ad uno stato precedente fa sì che la macchina possa essere esposta nuovamente a vulnerabilità che erano state precedentemente corrette;

- la possibilità di migrare una macchina virtuale su una o più macchine fisiche, rende più difficile la definizione del *Trusted Computing Base* [18] che, per una macchina virtuale, comprende anche tutte le macchine reali su cui la macchina è stata eseguita. È quindi più difficile tracciare la storia completa della macchina virtuale e capire quale sia il livello di fiducia che ad essa si può associare;

- la migrazione di una macchina virtuale fa sì che alcuni dati sensibili, come file, chiavi di cifratura, password etc, che risiedono sulla macchina virtuale, possano essere memorizzati su più host. Questo pone ovviamente dei problemi di riservatezza delle informazioni più elevati rispetto al caso in cui i dati risiedano sempre su una stessa macchina fisica.

1.5 Strumenti software

I software di virtualizzazione disponibili al momento sono molteplici. Nella tabella 1.2 ne sono elencati alcuni tra i più rappresentativi: per ognuno di essi si indicano le modalità di virtualizzazione e i sistemi operativi supportati.

Tra questi software, merita particolare attenzione la tecnologia di virtualizzazione proposta da Xen, che verrà discussa e analizzata nel capitolo seguente.

Capitolo 2

Xen

Contenuto

Xen [28] è un Virtual Machine Monitor *open source* sviluppato all'Università di Cambridge che adotta una strategia di paravirtualizzazione, e che quindi supporta sistemi operativi "guest" modificati. Xen permette di virtualizzare le risorse hardware di un computer e di farle condividere in maniera dinamica tra i sistemi operativi che sono in esecuzione sopra di esso, come mostrato in fig. 2.1. Al momento Xen è arrivato alla release 3.0.2-2.

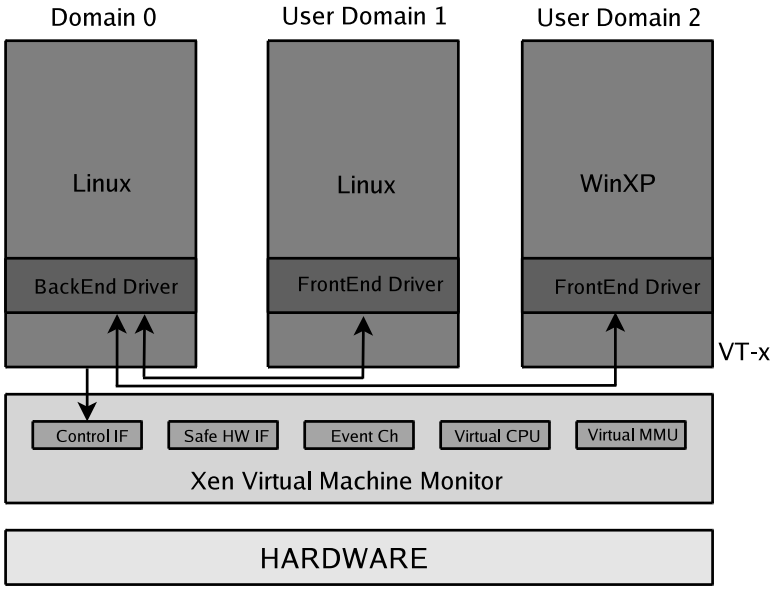


Figura 2.1: L'architettura di Xen 3.0

Nella terminologia Xen, le macchine virtuali sono chiamate *domini*. Xen fornisce degli ambienti di esecuzione isolati per i domini, tali per cui non è possibile che le attività di un dominio interferiscano con le attività di un altro dominio.

Xen richiede che i sistemi operativi siano modificati per poter essere eseguiti sul monitor tramite la tecnica della paravirtualizzazione. L'aspetto positivo della modifica ai sistemi operativi è una performance che si avvicina a quella nativa [31, 40].

Tre sono i requisiti che l'architettura di Xen vuole soddisfare:

a) garantire l'isolamento tra i domini: l'esecuzione di un dominio non deve influire sulle prestazioni di un altro dominio;

b) possibilità di eseguire su Xen molti dei sistemi operativi moderni;

c) l'overhead introdotto dalla virtualizzazione deve essere estremamente ridotto.

Molti sono i fattori che hanno portato gli sviluppatori di Xen a non percorrere la strada della virtualizzazione completa: uno di questi è che l'architettura x86 non facilita la virtualizzazione. Ad esempio, uno dei compiti più difficili è quello di virtualizzare l'MMU. Una soluzione a questo problema è quello offerto da VMware ESX Server [26], che utilizza la tecnica del "binary translation": però questa soluzione genera un overhead troppo elevato per garantire livelli di performance accettabili. Infatti, alcune operazioni che vengono eseguite frequentemente, quali

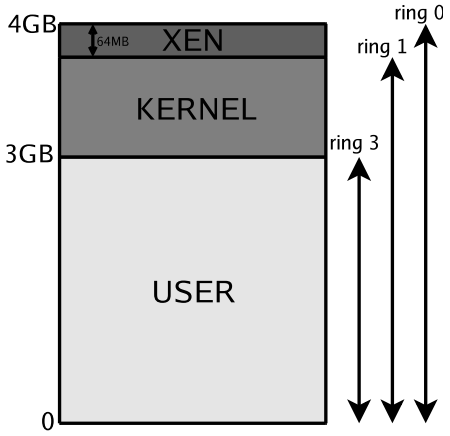


Figura 2.2: Lo spazio di indirizzamento nell'architettura x86/32

l'aggiornamento della tabella delle pagine, risultano particolarmente onerose usando questa tecnica.

È da sottolineare il fatto che, per poter eseguire un sistema operativo su Xen, le modifiche devono essere effettuate solo al codice del sistema operativo, mentre invece non sono richieste modifiche alle applicazioni scritte per quel sistema operativo.

2.1 L'interfaccia della macchina virtuale

L'interfaccia para-virtualizzata dell'architettura x-86, può essere suddivisa in tre componenti principali: la gestione della memoria, la CPU e i dispositivi di I/O.

2.1.1 Gestione della memoria

Tra i problemi che gli sviluppatori si Xen hanno dovuto affrontare, il più difficile è senz'altro quello di virtualizzare la memoria. La gestione hardware del TLB, ad esempio, richiede che per ottenere delle buone prestazioni:

1. i sistemi operativi "guest" siano responsabili della gestione della tabella delle pagine. Xen interviene solamente per garantire che le operazioni di aggiornamento eseguite siano corrette;

2. Xen risieda nello spazio di indirizzamento di ogni processo in esecuzione. In particolare, come mostrato in fig 2.2, esso risiede nei 64 MB della parte alta dello spazio di indirizzamento, nell'architettura x86/32, in modo da non di invalidare il TLB ad ogni cambio di contesto con l'hypervisor.

Tabella	Funzionalità	Dimensione
machine-to-physical table	tabella globale, in sola lettura, gestita da Xen: mappa le pagine della "machine memory" a quelle della "pseudo-physical memory"	dimensione proporzionale alla memoria fisica
physical-to-machine table	tabella associata ad ogni dominio: esegue la mappatura inversa	dimensione proporzionale alla memoria allocata al dominio

Tabella 2.1: Tabelle per la mappatura degli indirizzi su Xen

Così facendo, ogni volta che un sistema operativo "guest" richiede una nuova tabella delle pagine, ad esempio perché deve essere creato un nuovo processo, esso alloca una pagina della sua memoria per la nuova tabella e la registra con Xen. A partire da questo momento, tutti gli aggiornamenti a questa pagina, sulla quale il sistema operativo ha solo diritti in lettura, vengono eseguiti e validati da Xen.

Ovviamente, la regione alta della memoria da 64 MB riservata per Xen non è accessibile o rimappabile dai sistemi operativi. Questa funzionalità viene implementata usando la tecnica della segmentazione.

Un altro problema è dovuto al fatto che molti sistemi operativi richiedono che le pagine fisiche siano contigue, ma Xen non garantisce di allocare spazi di memoria fisica contigua per i domini. Per risolvere questo problema sono stati realizzati due diversi spazi di indirizzamento: uno contenente gli indirizzi della *machine memory* l'altro contenente quelli della *pseudo-physical memory*.

2.1.1.1 Machine memory e 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 dare l'illusione ai sistemi operativi di avere uno spazio di memoria contiguo anche se esso in realtà è sparso.

Per rendere possibile questa astrazione, esiste una tabella globale che mappa gli indirizzi appartenenti alla "machine memory" a quelli della "pseudo-physical memory", e una tabella in ogni dominio per implementare la mappatura inversa, come descritto nella tab. 2.1.

2.1.2 La CPU

Abbiamo visto nel capitolo precedente che i sistemi operativi possono eseguire tutte le istruzioni messe a disposizione dall'architettura x86, sia quelle privilegiate che tutte le altre. I livelli di privilegio, detti anche *rings*, nell'architettura x86 sono quattro, a partire dallo 0, con più privilegi, fino al ring 3, con minor privilegi. Solitamente il codice del sistema operativo viene eseguito nel ring 0, l'unico che può eseguire le istruzioni privilegiate, mentre le applicazioni vengono eseguite nel ring 3.

Visto però che solo Xen deve poter eseguire le istruzioni privilegiate, i sistemi operativi devono essere modificati così da poter essere eseguiti nel ring 1. In questa maniera, tutte le istruzioni privilegiate vengono para-virtualizzate e vengono eseguite da Xen. Se un sistema operativo tenta di eseguire un'istruzione privilegiata, il processore non la esegue e genera un'eccezione che trasferisce il controllo a Xen.

Per quanto riguarda eccezioni quali le system call o i fault di pagina, viene fornita ad ogni dominio una *virtual IDT* che è la tabella delle eccezioni con i puntatori ai gestori eseguiti all'interno del dominio. Ogni gestore di eccezione, al momento della registrazione con Xen, viene controllato per far sì che il codice non specifichi il ring 0 per la sua esecuzione.

L'hypercall (vedi par. 2.3) `set_trap_table` viene invocata dai domini per registare la tabella dei gestori delle eccezioni con Xen.

2.1.3 I dispositivi di I/O

Xen offre due tipologie di driver per fornire l'astrazione dei dispositivi virtuali:

- il *frontend* driver, che viene eseguito nel dominio non privilegiato, quello utente;
- il *backend* driver, che viene eseguito in un dominio che ha accesso diretto all'hardware, ad esempio il dominio 0 oppure un *driver domain*, descritto nel par. 2.5.

Il "frontend" driver fornisce al sistema operativo "guest" un'interfaccia per accedere al dispositivo uguale a quella del dispositivo reale. Il driver riceve le richieste dal sistema operativo e, visto che non ha accesso al dispositivo, invia una richiesta al "backend" driver. Il "backend" driver ha la responsabilità di soddisfare le richieste di I/O. Esso deve controllare che le richieste rispettino la politica di sicurezza, dopodiché passerà le richieste al dispositivo hardware. Quando il dispositivo ha terminato il lavoro di I/O, il "backend" driver segnala al "frontend" che i dati sono pronti e, successivamente, il "frontend" lo segnala al sistema operativo.

Il "backend" driver deve anche verificare che le richieste siano corrette e che non consentano di violare i principi dell'isolamento.

I driver usano la memoria condivisa per soddisfare le richieste e inviare i dati di risposta, e utilizzano un *event channel* come canale di segnalazione, come descritto nel par. 2.4. Per trasferire i dati di I/O da e verso i domini, Xen utilizza degli anelli di descrittori (*descriptor rings*), che identificano blocchi contigui di memoria fisica allocata al dominio, implementati mediante la tecnica del produttore-consumatore.

2.2 Il dominio 0

Durante l'inizializzazione di Xen, viene creato un dominio iniziale, che deve essere sempre presente, detto *dominio* 0. Questo dominio è responsabile delle funzionalità di controllo e gestione di Xen. Xen fornisce, ad uso del dominio 0, un'interfaccia di controllo (*control interface*) che permette di:

- creare e distruggere i domini;
- specificare alcuni parametri di esecuzione per i domini. Ad esempio, la dimensione totale della memoria allocata;
- creare interfacce virtuali di rete;
- configurare alcuni aspetti di Xen: modalità di condivisione della CPU, regole di filtraggio dei pacchetti sulle interfacce virtuali di rete, etc.

La scelta di utilizzare un dominio "privilegiato" è dettata dall'obiettivo di separare il più possibile le politiche dai meccanismi.

2.3 Hypercall

L'interfaccia definita dalle *hypercall* è il meccanismo tramite cui i domini possono richiedere all'hypervisor di eseguire operazioni privilegiate: per fare questo viene generata una interrupt software, alla maniera delle chiamate di sistema. Questo meccanismo fa sì, tramite l'uso del vettore delle interruzioni "reale" presente in Xen, che venga invocata la procedura di esecuzione della relativa hypercall, definita dal registro `EAX`.

Sull'architettura x86/32, l'istruzione da eseguire per effettuare la trap a Xen è la `int $82`, che può essere eseguita dal ring 1, il livello di privilegio associato all'esecuzione dei sistemi operativi "guest" su Xen.

2.3.1 Esempio di hypercall: mmu_update

I sistemi operativi "guest" hanno solamente accesso in lettura alla tabella delle pagine: ogni volta che devono aggiornare una tabella o crearne una nuova, invocano l'hypercall:

```
mmu_update(mmu_update_t *req, int count, int *success_count)
```

Questa hypercall aggiorna la tabella delle pagine del dominio: viene richiesto all'hypervisor di effettuare "count" aggiornamenti; "success_count" riporta il numero di aggiornamenti effettuati con successo.

In Xen questo è il modo operativo di default. In alternativa, è anche possibile dare l'illusione ai sistemi operativi "guest" che la tabella delle pagine sia scrivibile. Al momento della scrittura in una pagina appartenente alla tabella delle pagine, viene eseguita una trap a Xen che rimuove temporaneamente la pagina dalla tabella delle pagine, controlla che l'operazione sia valida e "riconnette" la pagina, in un secondo tempo, in modo che possa essere usata dal sistema operativo [78].

Un'altra alternativa è l'uso della tabella delle pagine ombra (*shadow page table*), in cui ogni sistema operativo "guest" ha una sua copia della tabella delle pagine, di cui il livello firmware non è a conoscenza. È compito di Xen fare sì che le due tabelle siano mantenute aggiornate, modificando la tabella ombra quando la tabella delle pagine "reale" viene modificata e viceversa [78].

2.4 Event channel

Le interruzioni vengono virtualizzate in Xen mappandole con dei relativi *event channels*, che forniscono un meccanismo per la trasmissione asincrona di notifiche ai domini. I domini associano ad ognuno degli eventi una funzione "callback" tramite l'hypercall `set_callbacks`; Xen ha il compito di stabilire, per ogni interruzione, a quale dominio essa è indirizzata [78].

Tra le notifiche di eventi che possono essere effettuate ai domini c'è anche, ad esempio, la richiesta di terminazione di un domino.

Per ogni dominio viene creata una bitmap, con un bit associato ad ogni evento, che Xen si preoccupa di modificare ogni volta che deve essere notificato il sistema operativo di un evento. A questo punto, viene eseguita la procedura di gestione per quell'evento e viene resettato il bit corrispondente.

Un altro flag presente nella bitmap, associato ad ogni evento, specifica se l'evento corrispondente debba essere mascherato. Ciò è del tutto analogo alla disabilitazione delle interruzioni di un processore.

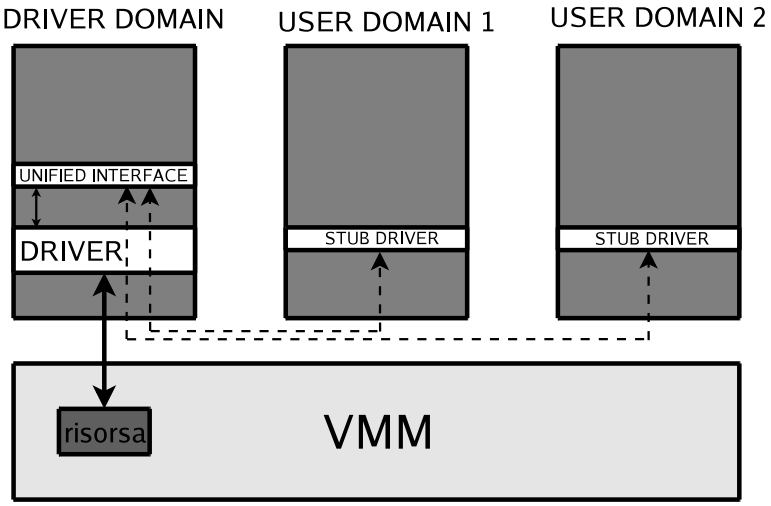


Figura 2.3: Il driver domain

2.5 Accesso sicuro all'hardware

Xen permette la condividisione dei dispositivi di I/O tra i vari sistemi operativi in esecuzione, isolando il codice dei driver dei dispositivi all'interno di domini. È infatti possibile eseguire i driver all'interno di domini opportuni, chiamati *driver domain*. Questi domini sono macchine virtuali create con lo specifico compito di contenere il codice dei driver e di permetterne la condivisione tra le varie istanze dei sistemi operativi [42].

Inoltre, grazie all'interfaccia che i driver hanno verso il livello hardware/firmware, che è chiamata *Safe Hardware Interface*, viene limitato l'accesso che i driver hanno verso le risorse hardware che gestiscono. Si ottiene altresì che i "crash" dei driver siano contenuti e abbiano ripercussioni minime sul sistema, perché sono limitati al dominio che li ospita.

Poiché in questa soluzione i driver sono eseguiti all'interno di macchine virtuali create apposta per ospitarli, viene esportata un'interfaccia unificata verso i driver che può essere usata contemporaneamente da più macchine virtuali. È invece possibile disporre di una sola implementazione dei driver, quella che è residente nel "driver domain" (vedi fig. 2.3).

Inoltre, un malfunzionamento dei driver può al più bloccare la macchina virtuale su cui è installato. Le altre macchine virtuali che accedono al driver tramite l'interfaccia unificata non saranno affette da questo problema.

2.5.1 L'architettura Safe Hardware

L'architettura di questo modello comprende tre aspetti principali (vedi fig. 2.4):

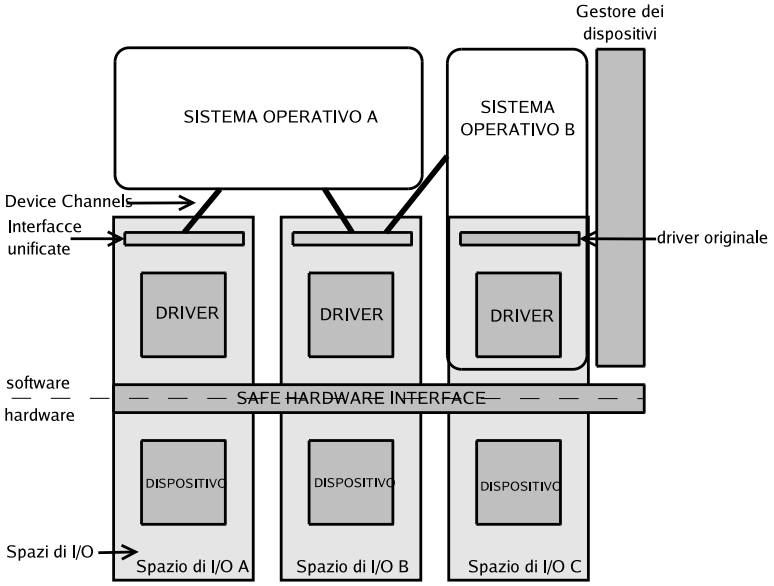


Figura 2.4: L'architettura "safe hardware"

- gli spazi di I/O;
- l'interfaccia unificata;
- un gestore dei dispositivi.

2.5.1.1 Gli spazi di I/O e l'isolamento

Nei sistemi operativi moderni, spesso i driver sono contenuti nel kernel. Quindi anche un "crash" di un driver può provocare l'arresto del sistema operativo. Questo è dovuto alla mancanza di isolamento tra il codice del driver e il resto del sistema operativo. Per risolvere questi problemi, Xen propone l'architettura "Safe Hardware", che ha i seguenti obiettivi:

- isolare l'esecuzione dei driver dalle altre componenti software del sistema;
- consentire ai driver l'accesso ai soli dispositivi che controllano;
- consentire solo operazioni sicure al dispositivo.

Per quanto riguarda il primo obiettivo, Xen installa il driver in un dominio separato. Come per tutti gli altri domini, anche questo dominio è isolato dagli altri mediante i normali meccanismi di protezione forniti dal VMM. Inoltre, un driver che presenta dei malfunzionamenti può essere riavviato senza impatti sulle altre parti del sistema.

Il secondo obiettivo, è raggiunto grazie al concetto di *I/O Spaces*. Questo è un meccanismo di protezione che prevede di assegnare delle specifiche risorse fisiche per l'interazione tra il driver e il dispositivo, ad esempio la memoria, i registri del dispositivo, gli interrupt, vincolando il driver ad accedere solamente alle risorse assegnategli.

Per ottenere l'ultimo obiettivo, Xen può prevenire operazioni potenzialmente "pericolose" da parte dei dispositivi, quali l'accesso in DMA ad indirizzi arbitrari o l'accesso diretto ad altri dispositivi, solo mediante un opportuno supporto a livello hardware/firmware [11]. Ad esempio, con la tecnologia IOMMU [35, 36], che è usata per tradurre indirizzi appartenenti allo spazio di I/O a indirizzi appartenenti allo spazio della macchina, è possibile limitare l'accesso in DMA dei dispositivi a parti di memoria che appartengono ad altri domini, ad esempio in quelli in cui non risiede il driver associato al dispositivo. In questo scenario, Xen si occupa di aggiornare e validare gli accessi alla IOMMU.

2.5.1.2 Interfacce unificate

Questa architettura definisce un insieme di interfacce di alto livello, ognuna per ogni classe di dispositivo. In questo caso, gli sviluppatori dei sistemi operativi, o i produttori dei dispositivi di I/O, devono solamente implementare dei "piccoli" driver per ognuna di queste classi di dispositivi, che hanno il compito di dialogare con l'interfaccia unificata.

La comunicazione tra i sistemi operativi e le interfacce unificate avviene attraverso l'uso dei *device channel*, collegamenti punto-punto tramite cui i sistemi operativi e il "driver domain" possono scambiarsi messaggi in maniera asincrona. Questi canali vengono inizializzati con l'aiuto del system controller [42].

2.5.1.3 Il gestore dei dispositivi

Il gestore dei dispositivi ha il compito di fornire un insieme di interfacce di controllo uniforme per tutti i dispositivi, in maniera analoga a quanto svolto dal BIOS su un normale computer.

2.5.2 Gestione delle interruzioni

La gestione di un'interruzione viene implementata mediante una routine interna a Xen, prima di passare il controllo al dominio in cui risiede il driver che gestisce il dispositivo che ha generato l'interruzione. Questo permette a Xen di controllare il sistema e di schedulare a suo piacere il "driver domain" per permettergli così di eseguire la routine di servizio dell'interruzione.

Quando il dominio verso cui l'interruzione è indirizzata viene schedulato, esso riceverà la notifica dell'interruzione tramite l'uso degli "event channel". Ogni dominio può avere al

massimo 1024 “event channel”, e ognuno di questi è formato da due bit: il primo usato per segnalare l’evento, il secondo per mascherarlo.

2.5.3 Memoria condivisa

Lo scambio di messaggi tra un dominio e un altro, viene implementato mediante la tecnica della memoria condivisa. Un dominio che vuole condividere una pagina della sua memoria con un altro deve presentare a Xen una *grant reference*, un indice a una tabella interna al dominio (la *grant table*) contenente tuple. Ogni tupla ognuna indica:

- la pagina da mappare;
- il dominio a cui si vuole consentire la mappatura della pagine nel proprio spazio di indirizzamento;
- i diritti di mappatura, ad esempio se essa è in sola lettura.

Quando un altro dominio richiede di mappare nella propria memoria una pagina condivisa, esso deve presentare una “grant reference”, che ha la stessa funzione di una *capability*. Infatti, tramite essa Xen può verificare, usando una sua tabella interna privata (la *active grant table*), o la “grant table” del dominio di cui si richiede la pagina per la condivisione, se il dominio richiedente ha il diritto per mappare la pagina nel proprio spazio di indirizzi.

2.6 Il modulo per il controllo degli acessi

Recentemente il progetto sHype [71] è stato integrato in Xen per fornire un insieme di funzionalità aggiuntive, tra cui:

- controllo delle risorse;
- politiche di controllo degli accessi tra le macchine virtuali;
- isolamento delle risorse virtuali.

Il modulo per il controllo accessi è pensato per permettere di definire politiche MAC. Tali politiche sono definite a livello di sistema invece che dagli utenti.

L’architettura di questo progetto prevede tre componenti, illustrate in fig. 2.5:

a) il **policy manager**: si occupa di definire la politica;

b) i **mediation hooks**: controllano l’accesso dei domini alle risorse condivise. Il controllo è basato sulle decisioni fornite tramite meccanismi di callback (vedere dopo);

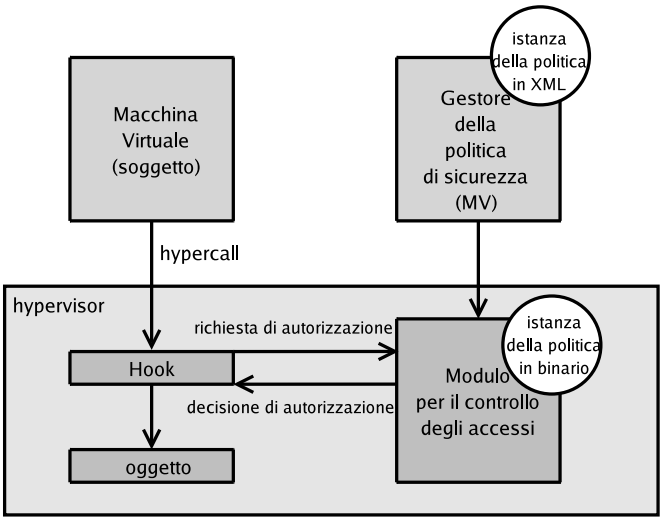


Figura 2.5: Il modulo per il controllo degli accessi

c) l'**access control module**: è responsabile di eseguire le callback dopo aver preso la decisione sulla base della politica di sicurezza corrente.

Gli obiettivi che il progetto sHype si prefigge per la sua integrazione in Xen sono duplici:

- garantire al codice un significativo livello di sicurezza (Common Criteria EAL4 [5]), effettuando solo cambiamenti minimi al codice del VMM;
- l'overhead dovuto ai controlli di accesso deve essere molto basso.

Di seguito sono analizzati i componenti principali di questa architettura.

2.6.1 Il reference monitor

Il componente centrale dell'architettura per il controllo degli accessi è il cosidetto *reference monitor* che provvede ad isolare le macchine virtuali (caso di default) e, quando richiesto, permette di condividere le risorse secondo la politica MAC stabilita.

Come già detto al par. 1.4.1.1, sono tre i requisiti principali di ogni reference monitor. Esso deve:

a) mediare tutte le operazioni critiche per la sicurezza;

b) essere in grado di proteggersi da modifiche;

c) essere di dimensioni "piccole" in modo che sia possibile validarne la corretta implementazione.

Gli ultimi due requisiti sono garantiti dall'architettura del VMM, che è di dimensioni ridotte, in numero di istruzioni, ed è protetto dalle macchine virtuali in esecuzione grazie alla sua modalità di esecuzione. Per quanto riguarda il primo punto, sono ritenute critiche per la sicurezza tutte quelle operazioni che, per essere portate a termine, richiedono l'autorizzazione del sistema, il quale concede l'autorizzazione in accordo alle politiche MAC.

Le risorse che vengono controllate sono le risorse virtuali perché solamente queste risorse possono essere condivise, e la condivisione è possibile solo nei seguenti casi:

1. risorse virtuali presenti nell'hypervisor ("event channel" e "shared memory");
2. risorse virtuali locali tra le macchine virtuali. Ad esempio, risorse condivise tramite i *device domain*, i domini che gestiscono direttamente una risorsa;
3. risorse condivise tra più hypervisor. Ad esempio, nel caso del reference monitor distribuito descritto nel par. 1.4.1.1.

Per quanto riguarda il primo caso, è l'hypervisor stesso a garantire la corretta implementazione della condivisione. Nel secondo caso, invece, il modulo per il controllo accessi ha il compito di applicare i meccanismi di protezione, definiti dalla politica MAC corrente, tra domini appartenenti a coalizioni multiple. Oppure, nel caso delle risorse virtuali condivise tramite i "device domain", una possibile soluzione è quella per cui i domini stessi devono essere in grado di isolare correttamente la risorsa virtuale da condividere. Ad esempio, facendo eseguire in un "device domain" un *secure microkernel*. In questo caso è necessario che questi domini abbiano lo stesso grado di fiducia del reference monitor. Infine, nel caso di risorse condivise tra domini residenti su differenti hypervisor, si possono utilizzare tecniche di bridging, come quelle descritte nel par. 1.4.1.1.

2.6.2 Politiche di controllo accessi

Attualmente Xen implementa due politiche di controllo accessi: la Chinese Wall Policy e la Simple Type Enforcement Policy.

2.6.2.1 Chinese Wall Policy

Questa politica [38] permette di definire quali macchine virtuali non devono essere mappate concorrentemente sulla stessa macchina fisica: per fare ciò, vengono definiti un insieme di tipi (*ChWall-Type*) che sono assegnati alle macchine virtuali. Ogni tipo è implementato mediante un'etichetta diversa. Ad esempio, un dominio che si occupa di eseguire applicazioni di Home Banking avrà un'etichetta diversa da uno che esegue applicazioni per Internet.

Vengono inoltre specificati degli insiemi di tipi che sono in conflitto tra di loro (*ChWall-Conflict Set*), per cui i domini etichettati con tipi appartenenti allo stesso "ChWall-Conflict Set" non verranno mai eseguiti allo stesso tempo sul medesimo hypervisor.

2.6.2.2 Simple Type Enforcemente Policy

Questa politica permette di specificare quali sono le macchine virtuali che possono condividere informazioni e le macchine virtuali con cui possono condividerle. Questa politica controlla:

- le comunicazioni tra i domini. Tali comunicazioni possono avvenire tramite traffico di rete, eventi o memoria condivisa;
- l'accesso dei domini alle risorse fisiche.

Questa politica permette la comunicazione tra quei domini che hanno un medesimo *STE-type*. Un dominio, inoltre, può accedere ad una risorsa se il dominio e la risorsa condividono un tipo in comune.

Si possono avere due tipologie di domini con questa politica:

1. domini con un unico "STE-type": la correttezza dello scambio di informazioni tra tali domini viene garantita dai controlli di accesso eseguiti dall'hypervisor; è quindi indipendente dai domini e non richede la loro cooperazione;
2. domini che hanno più di un "STE-type", ad esempio un dominio che virtualizza una risorsa (l'hard-drive) per creare diverse istanze di quella risorsa (virtual block drives) che vengono utilizzate da altri domini. In questo caso, viene assegnato al "device domain" un "STE-type" che gli permetta di accedere alla risorsa fisica e, alle risorse logiche, si assegna lo "STE-type" analogo a quello dei domini che vogliono accedere alle risorse logiche. In questo caso, viene delegato al "device domain" l'attuazione della politica di controllo degli accessi.

2.6.3 Il Policy Manager

Per configurare le politiche MAC esiste il modulo *policy manager* che ha il compito di salvare le istanze delle politiche in un formato comprensibile agli utenti e di tradurre questo formato in un formato binario direttamente interpretabile dal modulo per il controllo degli accessi.

2.6.4 I Security Enforcement Hooks

Quando le macchine virtuali condividono le risorse, il reference monitor media gli accessi alle risorse inserendo dei *security enforcement hooks* nel codice dell'hypervisor. Ciò permette di garantire che il modulo per il controllo degli accessi sia invocato ogni volta che è necessario prendere una decisione sulla base della politica corrente per concedere, oppure negare, l'autorizzazione.

Il modulo per il controllo degli accessi prende la decisione sulla base della politica corrente e delle etichette assegnate alle macchine virtuali (i "ChWall-Type" e gli "STE-type") e alle risorse (gli "STE-type").

I "security enforcement hooks" devono raccogliere tutte le informazioni per il controllo degli accessi quali, ad esempio, le etichette delle macchine virtuali, delle risorse virtuali e il tipo di operazione richiesta. A questo punto richiederanno la decisione al modulo per il controllo degli accessi e applicheranno la decisione stabilita dal modulo.

I punti in cui vengono inseriti questi "security hooks" nell'hypervisor sono tre:

1. durante le operazioni di gestione dei domini, ad esempio la creazione, la distruzione, il salvataggio, il ripristino o la migrazione dei domini. Vengono inviate al modulo per il controllo degli accessi tutte le informazioni relative all'operazione richiesta, al dominio che richiede l'operazione e a quello su cui deve essere effettuata l'operazione. In questi casi, il modulo per il controllo degli accessi effettua delle callback per assegnare o rimuovere le etichette ai domini creati o distrutti, oppure per controllare il "ChWall-Type" assegnato ai domini prima di concedere l'autorizzazione per effettuare l'operazione;

2. durante le operazioni di setup e distruzione degli "event channel". In questo caso, prima di concedere o negare l'autorizzazione, il modulo per il controllo degli accessi verifica se i domini che creano il canale siano membri di una stessa coalizione;

3. per operazioni di setup, grant, rimozione dell'accesso sulla memoria condivisa. Il modulo per il controllo degli accessi si comporta come nel caso delle operazioni sugli "event channel".

Nel secondo e nel terzo caso viene utilizzata anche una tecnica di *caching* delle decisioni, in maniera tale che dopo il primo controllo, purché la politica di sicurezza corrente non sia stata cambiata, non sia necessario invocare il modulo per il controllo degli accessi per fargli prendere la stessa decisione, evitando così degli overhead non necessari.

2.7 L'estensione di Xen con IVT-x

Usando la tecnologia IVT-x è possibile creare domini sui quali vengono eseguiti sistemi operativi non modificati. Tali domini prendono il nome di *Hardware-Based Virtual Machine* (HVM) [81].

Il *control panel* è stato modificato ed esteso per permettere di creare, controllare e distruggere anche tali domini. Nella fase di creazione di una HVM, viene caricato il firmware per il sistema operativo "guest" (il BIOS), che non ha accesso alle risorse fisiche bensì a quelle virtuali: come BIOS è stato utilizzato BOCHS [3].

Il *Virtual CPU Module* si occupa di fornire l'astrazione della CPU al dominio HVM. Tra i suoi compiti, c'è quello di salvare lo stato del processore ogni volta che il dominio rilascia una CPU reale. Inoltre, il modulo ripristina lo stato ogni volta che il dominio viene schedulato e gli viene nuovamente assegnato il processore. Per ogni CPU virtuale in un dominino HVM, viene creata una struttura VMCS.

Il *Virtual MMU Module* si occupa di fornire al dominio l'astrazione della MMU. Dato che i "guest" vedono solo indirizzi pseudo-fisici, questo modulo deve tradurre questi indirizzi negli indirizzi di macchina. Viene creata una "shadow page table" per ogni dominio HVM che viene modificata alla stessa maniera della "guest page table", e viceversa.

Il modulo per la virtualizzazione dei dispositivi di I/O (il *Device Module*) si occupa di creare delle istanze di *device emulation model* nel dominio 0 per ogni dominio HVM, al fine di fornire l'astrazione per tutti i dispositivi di I/O della macchina reale. Alcuni "model" sono inclusi nell'hypervisor per aumentare le prestazioni, come per il *Programmable Interrupt Controller*. Il "device module" attende un evento da un HVM e, successivamente, lo invia al "device emulation model" appropriato. Quando questo avrà eseguito l'operazione di I/O, il "device module" ritornerà il risultato al dominio. Anche in questo caso, esiste la coppia "frontend" driver, nel dominio HVM, e "backend" driver nel dominio 0.

Capitolo 3

I sistemi di rilevamento delle intrusioni

Contenuto

Un sistema di rilevamento delle intrusioni (IDS: *Intrusion Detection System*) è un sistema automatico che ha il compito di rilevare tentativi di intrusione e attacchi ad un sistema informatico. Il suo scopo principale è quello di rilevare e segnalare usi non autorizzati, utilizzi impropri

e abusi di un sistema informatico sia da parte di utenti autorizzati che da intrusi. Esso deve essere in grado di rilevare le intrusioni in tempo reale e in maniera accurata: in particolare deve minimizzare il numero di *falsi positivi* [32], quando viene erroneamente segnalato un attacco, e di *falsi negativi*, quando un attacco non viene rilevato.

3.1 Obiettivi

Un sistema di rilevamento delle intrusioni è un programma software o un sistema *appliance* (un sistema integrato hardware e software) che automatizza il processo di rilevamento delle intrusioni. Un sistema per la prevenzione delle intrusioni (IPS: *Intrusion Prevention System*, vedi par. 3.3.5) è un sistema che, oltre ad eseguire gli stessi compiti di un IDS, cerca di bloccare i tentativi di intrusione prima che questi compromettano il sistema.

Lo scopo principale di un IDS è quello di rilevare incidenti prima che essi causino un impatto al sistema stesso. Ad esempio, gli IDS possono rilevare quando un attaccante ha compromesso un sistema dopo averne sfruttato una vulnerabilità [59].

Gli IDS possono essere configurati anche per rilevare comportamenti che violano le politiche di sicurezza: in questo senso, gli IDS agiscono in maniera simile ai *firewall* e riconoscono il traffico che non rispetta certe regole.

Inoltre, gli IDS possono essere utilizzati per:

- duplicare i controlli effettuati dal firewall. Ad esempio, se l'IDS nota del traffico sulla rete che il firewall avrebbe dovuto bloccare, ma che è comunque riuscito ad entrare nella rete interna, ad esempio per un errore di configurazione del firewall, l'IDS può ripetere gli stessi tipi di controlli;
- fornire funzioni di *logging* per documentare gli attacchi e i tentativi di intrusioni effettuati sulla rete;
- notificare gli amministratori della rete, tramite un *alert* (allarme), ogni volta che si verificano eventi meritevoli di attenzione.

Gli IPS possono inoltre:

- bloccare l'attacco. Azioni da eseguire per questo scopo sono: terminare una connessione di rete, bloccare il traffico destinato/inviato a/da un dato IP e/o porta;
- modificare le politiche di sicurezza quando viene rilevata una minaccia per la rete. Possono, ad esempio, aggiungere al firewall una regola per bloccare traffico ritenuto sospetto;

- modificare il contenuto del pacchetto di rete. Ad esempio, gli IPS possono normalizzare il traffico di rete, per cui il *payload* del pacchetto viene incapsulato in un nuovo header per cercare di prevenire attacchi basati su header malformati, o rimuovere allegati infetti, come virus allegati a e-mail.

3.2 Principi

Secondo il modello di Denning [37], per sfruttare le vulnerabilità di un sistema al fine di comprometterlo, il sistema deve essere usato in maniera anomala, cioè devono essere eseguite azioni che:

- si differenziano dalle normali azioni eseguite sul sistema;
- possono condurre a intrusioni/attacchi;
- sono inconsistenti con le specifiche di programmi privilegiati, critici per la sicurezza del sistema.

A questi tre principi sono associati altrettanti metodi per il rilevamento delle intrusioni: l'*anomaly detection*, il *misuse detection* e lo *specification-based detection*.

3.2.1 Anomaly detection

Questa metodologia assume che gli attacchi generino dei comportamenti diversi da quelli che vengono osservati in un sistema nel suo utilizzo normale. È quindi possibile individuare gli attacchi confrontando il comportamento attuale del sistema con quello ritenuto normale e segnalando ogni possibile deviazione da tale comportamento.

Per stabilire quale sia il comportamento "normale" del sistema, devono essere creati i cosidetti *profili* del sistema. Essi tengono traccia delle caratteristiche tipiche del sistema, quali gli utenti, i processi, le connessioni di rete e così via, registrando le attività che avvengono durante il normale funzionamento del sistema. I profili vengono creati monitorando le caratteristiche di interesse del sistema per un certo periodo di tempo.

I profili possono essere statici o dinamici: i primi vengono costruti nel periodo di *training*, il periodo iniziale nel quale vengono monitorate le caratteristiche tipiche del sistema, e non sono modificati successivamente. I profili dinamici, invece, possono variare col tempo, e quindi comportamenti ritenuti anomali in un certo periodo possono essere considerati normali in un altro e viceversa.

Il vantaggio principale della "anomaly detection" deriva dal fatto che, per rilevare le intrusioni, non c'è bisogno *a priori* di avere una base di conoscenza contenente le firme di tutti i

possibili attacchi che possono essere effettuati contro il sistema. Le difficoltà maggiori, invece, sorgono nel momento in cui si tratta di determinare le soglie di scostamento dal comportamento normale del sistema tali per cui un comportamento viene considerato anomalo oppure no.

Denning [39] suggerisce di basarsi su metriche e modelli statistici. Una *metrica* è il valore di una variabile casuale che rappresenta una misura quantitativa ottenuta durante un periodo di tempo. I tipi di metrica definiti sono tre:

1. numero di occorrenze nei file log che soddisfano una certa proprietà per periodo di tempo. Ad esempio, il numero di tentativi falliti di login effettuati in un arco temporale prefissato;

2. tempo intercorso tra due eventi in relazione tra loro. Ad esempio, tempo trascorso tra due login dello stesso utente;

3. misura dell'utilizzo delle risorse: questa metrica misura la quantità dell'utilizzo di una risorsa da parte di un soggetto durante un periodo temporale. Ad esempio, l'utilizzo del processore da parte di un programma durante la sua esecuzione.

A questo punto, data una metrica per una variabile casuale x e n osservazioni $x_1, \ldots x_n$, il compito dei modelli statistici per x è quello di stabilire se una nuova osservazione x_{n+1} è anomala rispetto le precedenti osservazioni.

In seguito sono descritti tre modelli presenti in [39].

3.2.1.1 Modello della soglia

Questo modello stabilisce valori minimi e massimi per il valore della variabile x: si è in presenza di un comportamento anomalo se nell'arco temporale in cui viene calcolato il valore di questa metrica, il valore della variabile x non è contenuto nei limiti fissati dalle due soglie.

3.2.1.2 Modello della media e della deviazione standard

In questo caso si conoscono la media e la deviazione standard della variabile x, ottenute dalle precedenti osservazioni: se il valore della nuova osservazione non cade all'interno dell'intervallo di confidenza [1] siamo in presenza di un comportamento anomalo.

3.2.1.3 Modello di Markov

Questo modello considera ogni evento come una variable di stato e l'esecuzione di un evento fa transire il sistema in un altro stato. Viene quindi generata una matrice di transizione tra stati

[1] dato un parametro d, l'intervallo di confidenza è uguale a $E[X] \pm d \times \sqrt{VarX}$: la probabilità che un valore cada fuori questo intervallo è al massimo $\frac{1}{d^2}$.

per caratterizzare la frequenza delle transizioni tra gli stati: siamo in presenza di un'anomalia ogni volta che in uno stato viene eseguito un evento che, tenuto conto della matrice e dello stato corrente, ha bassa probabilità di essere eseguito.

3.2.2 Misuse detection

Un sistema di rilevamento di intrusioni che utilizzi questa metodologia si affida ad una base di conoscenza contenente un insieme di firme che caratterizzano attacchi noti. Si analizzano le azioni che avvengono nel sistema e si confrontano con le firme appartenenti alla base di dati. Se esiste una corrispondenza tra i dati rilevati e una o più firme, siamo in presenza di un tentativo di intrusione.

Una firma (*signature*) rappresenta un pattern corrispondente ad un attacco noto, ad esempio:

- un tentativo di accesso via telnet o ssh con lo username "root";
- un allegato di mail con estensione `.exe`.

Lo svantaggio di questo metodo è che esso può identificare solo quegli attacchi che sono già noti. Ciò equivale a dire che la firma dell'attacco è già contenuta nella base di dati. La base di dati deve essere quindi frequentemente aggiornata. Inoltre, piccole variazioni nelle modalità di intrusione possono rendere diversa la firma di un attacco. Un tentativo d'intrusione può non essere rilevato come tale perché si discosta, anche se leggermente, dalla firma presente nella base di dati.

3.2.3 Specification-based detection

Questo metodo è speculare al "misuse detection": si affida a delle specifiche che descrivono qual è il comportamento atteso e corretto dei programmi critici per la sicurezza del sistema o come determinati protocolli di rete devono essere utilizzati. Durante il funzionamento del sistema, viene monitorato il comportamento dei programmi in esecuzione e del traffico di rete alla ricerca di azioni che violano queste specifiche.

Nel caso delle specifiche dei protocolli di rete, è noto che uno stesso protocollo può essere implementato in modo diverso dalla varie *software house* o dai produttori dei componenti hardware. Ciò implica che pattern di traffico che appartengono allo stesso protocollo possono apparire sotto forme diverse. Quindi l'IDS deve essere in grado di accettare variazioni dovute alle differenti implementazioni dello stesso protocollo.

Per semplificare l'analisi dell'IDS, in alcune reti il traffico viene "normalizzato", forzando una stessa implementazione in tutti quei casi in cui le specifiche ammettano implementazioni alternative del protocollo.

Il vantaggio di questo approccio è che esso può, come l'"anomaly detection", rilevare e segnalare anche attacchi non noti.

3.3 Classi di IDS

Questo paragrafo descrive le varie classi di IDS, che sono classificati in base al tipo di dati che analizzano, al meccanismo utilizzato per rilevare le intrusioni, alla tipologia di installazione che richiedono, ai sistemi in cui vengono installati e per altri fattori.

Tradizionalmente gli IDS sono suddivisi in Host IDS e Network IDS: a questi vanno aggiunti gli IDS ibridi, una combinazione dei due, i Wireless IDS e gli Intrusion Prevention System.

3.3.1 Host IDS

In questo caso, l'agente che si occupa di controllare il sistema risiede direttamente sull'host monitorato. Il meccanismo di base di questa tecnologia è il monitoraggio delle chiamate di sistema, dei file di log delle applicazioni e del sistema e dei record di audit. Questi file contengono informazioni su eventi relativi alla sicurezza del sistema, ad esempio tentativi di login o modifiche alla politica di sicurezza del sistema. L'agente ha quindi accesso diretto all'host che sorveglia e sorveglia unicamente questo host.

Tra i vari compiti che svolge, l'Host IDS deve:

- controllare i processi in esecuzione per rilevare programmi modificati. Ad esempio, i programmi `ls`, `ps` spesso sono modificati da un attaccante per celare l'esistenza dei processi "maligni", *backdoor* e altro, installati dall'attaccante stesso;
- rilevamento di *buffer overflow*. L'IDS può ad esempio controllare se un processo cerca di eseguire istruzioni appartenenti a regioni di memoria in sola lettura;
- rilevare tentativi di attacco al sistema o quali utenti cercano di elevare i propri privilegi oppure ripetuti fallimenti nel processo di login;
- controllare i file di log alla ricerca di eventi sospetti;
- controllare l'integrità del file system. Ad esempio, l'IDS può periodicamente calcolare l'hash di file critici per il sistema e confrontarli con quelli dei file originali, alla ricerca di differenze. Eventuali differenze possono, ad esempio, essere dovute a *trojan* inseriti nei file;
- controllare le modifiche ai permessi e ai proprietari dei file;

- controllare l'utilizzo delle risorse da parte dei processi in esecuzione per prevenire attacchi di tipo *Denial of Service* (DOS).

Spesso, gli agenti per Host IDS usano la tecnica dello *shim* [59]: lo "shim" è un frammento di codice che viene inserito all'interno di codice già esistente per intercettare i dati nel punto in cui sarebbero passati da un modulo di codice ad un altro. In questa maniera, l'IDS può analizzare i dati per determinare se l'azione richiesta può o no essere permessa. Gli "shim" possono essere usati per analizzare il traffico di rete, l'attività del file system o le chiamate di sistema.

In questo caso, l'Host IDS rallenta certamente le attività dei processi e degli utenti. Infatti, il processo e/o l'utente rimangono "bloccati" fino a quando l'IDS non ha preso la sua decisione. Requisito fondamentale per questa classe di IDS è quello di avere delle prestazioni elevate.

3.3.2 Network IDS

A differenza del'Host IDS, il Network IDS è installato su un sistema separato dall'host che sta monitorando. Inoltre, può monitorare vari host che risiedono su uno stesso tratto di rete. Il meccanismo di base è il monitoraggio del traffico di rete mediante *sniffing* dei pacchetti: il Network IDS analizza tutto il traffico che passa sulla rete alla ricerca di firme di attacchi noti, oppure alla ricerca di traffico statisticamente anomalo.

I pacchetti prelevati vengono passati ad un *detection engine* che, tramite delle regole interne, decide se è in presenza di un tentativo di intrusione. Per fare questo, è necessario sia ricostruire i vari flussi, ad esempio le connessioni TCP, che ricomporre i pacchetti che vengono frammentati per effetto della *Maximum Transmission Unit*, la dimensione massima che un pacchetto può avere su una particolare tecnologia di rete.

Lo svantaggio del Network IDS è il fatto di non poter "bloccare" il flusso di pacchetti per non farli giungere all'host: cioè, anche in presenza di un attacco, non è possibile filtrare i dati che transitano sulla rete.

Solitamente un Network IDS è collegato ad uno switch tramite una *spanning/mirror port*, che è una porta su cui viene copiato tutto il traffico delle altre porte, per cui l'IDS riesce a vedere tutto il traffico di rete.

Nel caso di una rete a fibra ottica, il sensore è collegato direttamente alla rete tramite un *network tap*.

Un Network IDS può inoltre[59]:

- identificare gli host della rete: ad esempio, gli indirizzi MAC e IP degli host che sono sulla rete monitorata;

- identificare i sistemi operativi degli host. Ciò è possibile tramite la tecnica del *passive fingerprinting*, che analizza caratteristiche particolari nel formato degli header dei pacchetti che permettono di rilevare il sistema operativo da cui il traffico viene generato;
- identificare le versione delle applicazioni. Ad esempio, il banner di una connessione telnet permette di risalire alla versione del server telnet;
- identificare le caratteristiche della rete. Ciò permette di tracciare una mappa della rete, definendo parametri quali il numero di salti tra due host o la presenza di firewall o di router.

Inoltre, un Network IDS deve eseguire un'accurata azione di logging degli eventi e delle attività sospette rilevate dalla rete, registrandone:

a) la data e l'ora;

b) l'ID di sessione (ad esempio, un numero consecutivo assegnato ad ogni nuova connessione TCP);

c) il tipo di evento;

d) la severità dell'evento, cioè, il suo grado di pericolosità;

e) l'IP e porta sorgente e destinazione, e il protocollo di trasporto;

f) il numero di byte trasmessi su quella connessione;

g) i dati appartenenti al *payload*, ad esempio codici HTTP di *request* e *response*;

I tipi di eventi più importanti che un Network IDS riesce a rilevare sono [59]:

- *buffer overflow*, *format string attack*, tentativi falliti di login dovuti alla generazione automatica di password, trasmissione via rete di virus o di *malwar*e etc;
- *port scanning*, attacchi di SYN o basati su frammentazione di pacchetti;
- indirizzi IP *spoofed*, in cui viene modificato l'indirizzo sorgente di un pacchetto IP;
- presenza di *backdoor*;
- uso di applicazioni proibite dalle politiche di sicurezza della rete.

I problemi più importanti per un Network IDS sono:

1. il traffico cifrato pone un evidente problema per i Network IDS che ispezionano il traffico alla ricerca di firme. Questo problema è sempre più presente data la grande diffusione degli strumenti di *encryption*;

2. visto che un Network IDS è spesso collegato ad uno switch su una port *mirror* che replica verso l'IDS tutto il traffico che transita sulla rete, deve essere in grado di analizzare i pacchetti anche in presenza di un volume di traffico estremamente elevato. Ciò con ovvie ripercussioni sul livello hardware e firmware dell'IDS;

3. i Network IDS non sanno quale sarà l'impatto del traffico sull'host né se il destinatario accetterà parte o tutto del traffico di rete.

3.3.3 IDS ibridi

Gli IDS ibridi sono basati su un'architettura che prevede *agenti* installati sugli host, come nel caso degli Host IDS, e dei *sensori* che analizzano il traffico di rete, come nel caso dei Network IDS. Gli agenti e i sensori riportano i risultati delle loro analisi ad un sistema centralizzato che ha funzionalità di elaborare i dati ricevuti e di correlare i vari dati per capire se sono in corso attacchi.

C'è da tenere presente però, che possono nascere alcuni problemi nel momento in cui si cerca di far coesistere differenti approcci per il rilevamento delle intrusioni in un unico sistema. Ad esempio, il formato dei dati di output utilizzato su un sistema può essere diverso da quello utilizzato su un altro. In questi casi, è necessario aggiungere una componente di interfacciamento tra i vari sistemi, ad esempio un *Security Information and Event Management* (SIEM) che è uno strumento software in grado di ricevere e analizzare messaggi provenienti da differenti IDS per effettuare correlazioni tra i vari dati ricevuti.

3.3.4 Wireless IDS

Un *Wireless IDS* è un IDS che ha il compito di monitorare il traffico su una rete wireless per rilevare attività sospette inerenti l'uso dei protocolli. In particolare, un Wireless IDS può rilevare attacchi e violazioni alle politiche sull'uso dei protocolli esaminando le comunicazioni che avvengono sulla rete tramite i protocolli della famiglia IEEE 802.11.

I tipi di eventi monitorati da un Wireless IDS sono [59]:

- uso non autorizzato di apparati wireless: ad esempio, *access point* non autorizzati;

- stazioni o "access point" che usano protocolli non sicuri: ad esempio, stazioni che usano il protocollo *WEP* invece del *WPA* (o *WPA2*);

- pattern di traffico anomali: ad esempio, un numero elevato di stazioni che inviano il loro traffico solamente verso un "access point";
- uso di *war driving tools*, che sono scanner utilizzati dagli attaccanti per rilevare reti wireless "aperte", cioè quelle reti che permettono l'accesso senza autenticazione o tramite un meccanismo di autenticazione insicuro, come il WEP;
- attacchi di tipo *Denial Of Service*: ad esempio, l'invio di grandi quantità di dati verso un "access point";
- attacchi di *spoofing*: quando un attaccante cerca di impersonare l'identità di un utente legittimo.

I sensori Wireless possono essere sia fissi, ad esempio su un PC desktop con interfaccia wireless, che mobili, ad esempio un portatile con integrato un Wireless IDS. Questi ultimi sono particolarmente utili per individuare attaccanti in movimento.

3.3.5 Intrusion Prevention System

A differenza degli IDS, gli IPS usano meccanismi proattivi di difesa che in presenza di tentativi di intrusione/attacco possono bloccare il traffico prima che esso riesca effettivamente a causare un danno.

In pratica, gli IPS sono degli IDS che hanno l'ulteriore capacità di eseguire un'azione in risposta ad un attacco. Inoltre, gli IPS sono degli apparati *inline*. Nel caso di un Network IPS, ad esempio, i pacchetti intercettati saranno immessi nuovamente nella rete solo dopo aver eseguito tutti i controlli necessari.

In questo contesto, con IPS indichiamo esclusivamente i Network IPS. Come illustrato nella fig. 3.1(a) e nella fig. 3.1(b), i Network IDS non sono in grado di filtrare il traffico di rete, mentre i Network IPS possono farlo.

Un IPS può bloccare pacchetti in transito tra l'attaccante e l'host destinatario dei pacchetti oppure, ad esempio, può modificare il contenuto del traffico a livello di applicazione, per far sì che un attacco a questo livello non abbia successo.

Ci sono quattro tipi di contromisure che un IPS può applicare per prevenire un attacco [67]:

a) a livello di **data-link**: disattivare una porta dello switch da cui provengono gli attacchi in locale;

b) a livello di **rete**: interagire con il firewall, o il router, per aggiungere una regola di controllo degli accessi che blocchi il traffico da/verso uno specifico IP, o una specifica porta;

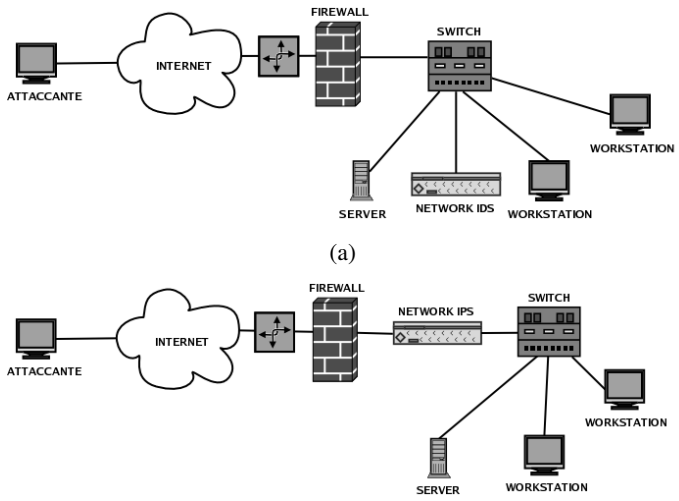


Figura 3.1: Network IDS (a) e Network IPS (b)

c) a livello di **trasporto**: generare pacchetti TCP RST per cercare di forzare la chiusura di una sessione;

d) a livello di **applicazione**: modificare i dati al fine di rendere inoffensivi attacchi eseguiti a questo livello.

3.4 Architettura

Le componenti principali di un'architettura IDS sono (fig. 3.2):

- l'**agent**, nel caso di un Host IDS, o il **sensor**, nel caso di un Network IDS. L'agent ha il compito di raccogliere informazioni dal sistema che controlla. Possibili sorgenti di informazione sono i file di log, l'analisi dei processi in esecuzione, le chiamate di sistema, il traffico di rete generato o destinato da/a quel sistema. Gli agent inviano le informazioni raccolte al director dopo aver svolto, se necessario, delle azioni di *preprocessing*, tra le quali la conversione dei dati in un formato standard, ad esempio in IDMEF (*Intrusion Detection Message Exchange Format*) [50]. Il director può interrogare gli agent per richiedere di inviare ulteriori dati. Nel caso di più agent installati sullo stesso host, può essere presente anche un **collector** che fa da tramite tra i vari agent e il director, eseguendo anche funzionalità di filtraggio e preprocessing. Il sensor ha il compito di analizzare il traffico di rete alla ricerca di intrusioni/attacchi, e di notificare il director in caso di eventi critici per la sicurezza del sistema;

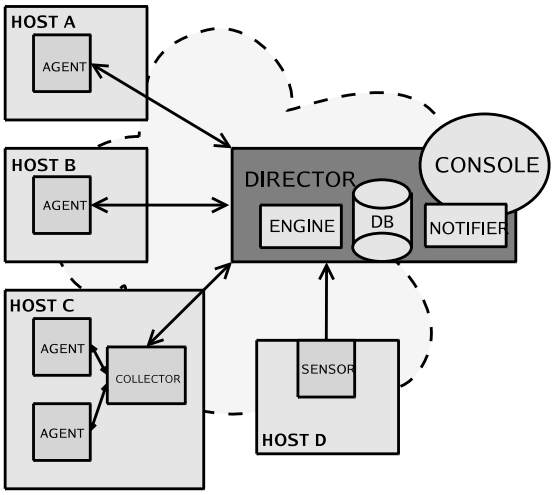


Figura 3.2: Tipica architettura di un IDS

- il **director**, che raccoglie tutte le informazioni che riceve dagli agent e dai sensor estrapolando informazioni utili, correlando i vari dati, alla ricerca di tentativi di intrusione. È installato su un sistema separato e centralizzato. Spesso, usa meccanismi ed algoritmi propri dell'intelligenza artificiale per modificare il proprio comportamento nel tempo e cercare così di ridurre il numero di falsi positivi generati;

- il **notifier**: una volta che il director ha rilevato un tentativo di intrusione o di attacco, il notifier ha il compito di scegliere la contromisura adeguata. Ad esempio, può avvertire il responsabile della sicurezza o fermare il traffico di rete;

- il **database server**, che contiene tutti gli eventi ricevuti dai sensori e dagli agenti;

- la **console**, che costituisce l'interfaccia degli amministratori per la configurazione dell'IDS. Può essere presente anche una console usata solamente per attività di monitoraggio e analisi, ad esempio dove vengono visualizzati i messaggi di log.

3.5 Esempi

Nei successivi paragrafi verrano presentati alcuni IDS sviluppati in ambito di ricerca o presenti sul mercato.

3.5.1 Analisi delle sequenze di chiamate di sistema

In [51] viene presentato un metodo per rilevare le intrusioni tramite l'analisi delle sequenze di chiamate di sistema nei processi privilegiati. Per ogni programma analizzato viene costruito un profilo che rappresenta il comportamento normale del programma: ogni deviazione da tale profilo viene considerata come anomalia. Gli eventi che vengono monitorati sono brevi sequenze di chiamate di sistema: non interessano i parametri passati alle chiamate, ma solo l'ordinamento temporale delle varie chiamate.

L'algoritmo utilizzato prevede che durante la fase di "training" venga costruita una base di dati analizzando tutte le tracce delle chiamate di sistema generate dal programma considerato, e salvando nella base di dati tutte le diverse sequenze di lunghezza k (per k fissato) che appaiono nella tracce. Dopo aver costruito la base di dati, è possibile monitorare il comportamento del processo che esegue il programma e confrontare il comportamento con le sequenze di chiamate di sistema presenti nella base di dati.

Ad esempio, se viene osservata questa traccia di chiamate di sistema:

```
open; read; mmap; mmap; open; read; mmap;
```

per $k = 3$ si ottengono queste sequenze uniche:

```
open; read; mmap;
read; mmap; mmap;
mmap; mmap; open;
mmap; open; read;
```

Durante il monitoraggio del processo si utilizza lo stesso algoritmo usato nella fase di "training", controllando le sequenze di lughezza k di chiamate di sistema generate dal processo da monitorare, e si ricercano tali sequenze nella base di dati. Le sequenze che non appaiono nel database sono dette *mismatches* (riscontri negativi). Più alto è il numero di "mismatches", più è alta la probabilità di essere in presenza di un comportamento anomalo.

Per determinare quanto due sequenze i e j distano l'una dall'altra, cioè quanto sono diverse, viene usata la *distanza di Hamming* $d(i, j)$. Questa distanza è uguale al numero di elementi differenti tra le due sequenze i e j. Per ogni nuova sequenza i, la distanza minima di Hamming $d_{min}(i)$ tra la sequenza i e l'insieme delle sequenze contenute nella base di dati è:

$$d_{min}(i) = min\{d(i, j) \text{ per tutte le sequenze normali j}\}$$

Ad esempio, supponendo che la traccia di un processo monitorato sia:

```
open; read; mmap; mmap; open; mmap; mmap;
```

si ottengono queste nuove sequenze uniche (rispetto le precedenti):

```
mmap; open; mmap;
open; mmap; mmap;
```

per cui ci saranno due riscontri negativi, che sono il 40% della traccia, e due d_{min} che hanno valore 1.

Per limitare il numero di falsi positivi, si può settare una soglia C tale per cui sia:

$$d_{min}(i) \geq C$$

con $1 \leq C \leq k$. In questo caso, vengono segnalate come anomale solo le sequenze che sono significativamente differenti dalle sequenze normali.

3.5.2 Monitoraggio dei processi

In [73] viene presentato un approccio per rilevare attacchi che si basa su delle specifiche che definiscono il comportamento permesso ai programmi. La specifica è definita mediante asserzioni logiche sulle sequenze di chiamate di sistema e sul valore degli argomenti delle chiamate.

Il punto di partenza di questo approccio è che è, teoricamente, possibile identificare tutti gli attacchi osservando le chiamate di sistema eseguite dai processi in esecuzione su un host, filtrando quelle chiamate che sappiamo aver conseguenze negative sulla sicurezza del sistema.

Viene usato un linguaggio di specifiche di alto livello, l'*Auditing Specification Language* (ASL), che viene utilizzato per specificare comportamenti normali e anormali dei processi tramite asserzioni logiche. Quando, a tempo di esecuzione, vengono riscontrate delle anomalie nel comportamento atteso per un processo, vengono iniziate azioni di difesa, anche queste specificate in ASL, per cercare di contenere l'attacco.

ASL utilizza i *general event patterns* per specificare i comportamenti validi o non validi. Un *atomic pattern* è definito come:

$$e(a_1, \ldots, a_n)|C$$

in cui e identifica un evento e C, invece, identifica un'espressione booleana valutata sulle variabili $a_1, \ldots, a_n$.

Ad esempio, per ottenere privilegi di root tramite una shell, viene spesso eseguita la chiamata `execve()` specificando `/usr/bin` come parametro. Nella specifica seguente, viene impedito al programma `fingerd`, che può essere affetto da una vulnerabilità di "buffer overflow", di

eseguire programmi arbitrari tramite la `execve()`, permettendogli comunque di eseguire i soli programmi necessari per il suo normale funzionamento:

```
execve(f) | f != ``/usr/ucb/finger'' -> exit(-1)
```

In questo caso, ogni volta che `fingerd` prova a eseguire la `execve()`, se il nome del file passato come primo argomento è diverso da `/usr/ucb/finger`, allora viene eseguita una `exit(-1)`.

Consideriamo un altro esempio in cui vogliamo essere certi che un programma scaricato da una fonte non sicura non procuri danni all'host su cui è in esecuzione. Di conseguenza, permettiamo al programma solamente l'accesso in lettura ai file leggibili da tutti e in scrittura solamente nella directory `/tmp`, e non permettiamo invece che esso possa eseguire altri programmi o che possa utilizzare la rete. Le specifiche ASL corrispondenti sono:

```
open(file, mode) |
  [(!inTree(realpath(file), ``/tmp'') &&
  (mode & (O_WRONLY|O_APPEND|
           O_CREAT | O_TRUNC))||
  !accessible(realpath(file), mode,
              ``nobody''))
-> fail(-1,EACCESS);

exec || connect || bind || chmod ||
 chown || chgrp || create ||
 truncate || sendto || mkdir
-> exit(-1);
```

In questa specifica viene utilizzata una funzione di supporto, `inTree`, che determina se un file appartiene alla directory passata come parametro, o ad una sua sottodirectory. Quindi, se il file viene aperto in modalità di scrittura e il file non appartiene alla directory `/tmp`, o una sua sottodirectory, oppure il file non è leggibile da tutti, viene restituito un codice d'errore e l'operazione non viene permessa. Inoltre, nel caso in cui vengano invocate chiamate di sistema non permesse, ad esempio le chiamate `bind`, `connect`, `sendto` che sono utilizzate per effettuare comunicazioni via rete, il programma viene forzatamente terminato.

3.5.3 Autonomus agents

In [34] un *autonomus agent* è definito come un agente software che esegue delle funzioni di monitoraggio per la sicurezza su un host. Gli agent sono autonomi nel senso che sono entità

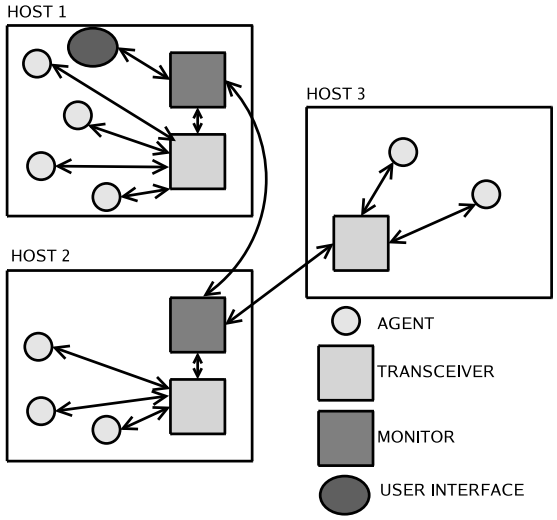


Figura 3.3: L'architettura AAFID

indipendenti tra loro.

Viene proposta l'architettura AAFID (*Autonomous Agents For Intrusion Detection*) che usa, come elemento di più basso livello, gli agent per collezionare e analizzare dati, e crea una struttura gerarchica tra questi agent. Altri componenti dell'architettura sono i *transceiver* e i *monitor*. I transceiver sono entità installate su ogni host e hanno funzioni di controllo verso tutti gli agent installati su quell'host. Ad esempio, possono avviare e terminare gli agent e inviare loro comandi per configurarli. I monitor sono entità che ricevono i risultati di monitoraggio dai vari transceiver installati sugli host. I monitor possono anche correlare i vari dati e scoprire intrusioni che avvengono su diversi host (vedi fig. 3.3).

3.5.3.1 Gli agent

Gli agent monitorano aspetti specifici di un host, segnalando comportamenti anomali o sospetti al transceiver che risiede sullo stesso host. Un agent non genera un allarme in modo autonomo, perché questo è compito dei transceiver o dei monitor dopo aver ricevuto vari messaggi dai diversi agent. Infatti, i transceiver, ricevendo informazioni dai vari agent installati su un host, hanno una visione più ampia dello stato dell'host, mentre i monitor hanno una visione globale della rete che stanno monitorando grazie alle informazioni che vengono inviate loro dai transceiver.

Agent diversi possono avere una complessità estremamente diversa. Ciò che interessa non è il diverso grado di complessità, ma il formato appropriato usato per scambiare informazioni con il transceiver che deve essere lo stesso per tutti.

3.5.3.2 I transceiver

I transceiver hanno il compito di comunicare i dati all'esterno dell'host. Essi hanno due ruoli: *controllo* e *data processing*. Nel primo caso, i transceiver possono:

- avviare e fermare gli agent che sono in esecuzione sullo stesso host del transceiver. Queste azioni possono avvenire su sollecitazione del monitor o in risposta a particolare eventi;
- tenere traccia degli agent che sono in esecuzione sull'host;
- rispondere alle richieste del monitor. Ad esempio, quando il monitor richiede ulteriori dati da uno specifico agent.

Nel ruolo di "data processing", il transceiver deve:

- ricevere e analizzare i dati ricevuti dagli agent in esecuzione sullo stesso host;
- eseguire attività di processing dei dati. Ad esempio, utilizzare il corretto formato dei messaggi per comunicare con il monitor;
- inviare le informazioni al monitor.

3.5.3.3 I monitor

La differenza fondamentale tra il monitor e i transceiver è che il monitor può controllare entità che risiedono su diversi host. Anche il monitor assume i due ruoli di controllo e di "data processing"; nel primo caso esso può:

- ricevere informazioni e istruzioni da altri monitor e controllare l'attività dei transceiver e/o di altri monitor;
- comunicare i risultati agli utenti tramita una console.

Invece, nel ruolo di "data processing" il monitor ha il compito di:

- ricevere le informazioni inviategli dai transceiver che controlla ed eseguire correlazioni di più alto livello rispetto a quanto possono fare i transceiver;
- rilevare attacchi che colpiscono più host.

3.5.4 Un sistema di rilevamento delle intrusioni distribuito

In [75] viene descritto il prototipo di un *Distributed Intrusion Detection System* (DIDS) in grado in monitorare non solo la rete ma anche i diversi host che essa connette.

Nell'architettura sono previsti un *DIDS director*, un *host monitor* per ogni singolo host, e un *LAN monitor* per controllare la rete intera. Tutte le informazioni raccolte dalle varie componenti vengono inviate ad un sistema centrale per essere analizzate. Gli host monitor e i LAN monitor sono responsabili di raccogliere dati e rilevare attività sospette, mentre il DIDS director ha il compito di valutare i dati ricevuti per decidere se siamo in presenza di un'intrusione.

3.5.4.1 L'host monitor

L'host monitor è composto da un *host event generator* (HEG) e da un *host agent*. L'HEG raccoglie e analizza i dati di log del sistema operativo alla ricerca di eventi "notevoli", ad esempio tentativi falliti di login, cambiamenti alla politica di sicurezza del sistema, accessi al sistema da remoto. Questi eventi vengono inviati al director per ulteriori analisi. Il compito dell'host agent è quello di gestire la comunicazione tra l'host monitor e il DIDS director.

3.5.4.2 Il LAN monitor

Il LAN monitor è composto da un *LAN event generator* (LEG) e da un *LAN agent*. Il LEG deve monitorare il traffico sulla rete e segnalare attività, quali connessioni *rlogin* o *telnet*. Il LAN agent comunica i dati al DIDS director.

3.5.4.3 Il DIDS director

Il DIDS director è composto da tre parti:

- il *communications manager*, che gestisce tutte le comunicazioni tra il director e gli host monitor e il LAN monitor;
- l'*expert system*, che riceve i dati dal *communications manager* e che deve valutare i dati ricevuti e analizzare lo stato della sicurezza del sistema monitorato. Grazie ai dati che riceve dai vari monitor è in grado di inferire fatti riguardanti la sicurezza dei singoli host e del sistema (la rete e gli host). È un sistema basato su delle regole con capacità di apprendimento;
- la *user interface*, che permette al responsabile della sicurezza di interagire con l'intero sistema. Ad esempio, per richiedere ulteriori informazioni ai monitor.

Il sistema esperto usato dal DIDS director è scritto in Prolog e le regole usate derivano dal modello gerarchico *Intrusion Detection Model* (IDM). Questo modello descrive in maniera astratta i dati usati per fare inferenze sugli attacchi. Esso, cioè, descrive come i dati ricevuti in maniera distribuita dagli host possono essere trasformati in ipotesi di alto livello su intrusioni e sullo stato di sicurezza del sistema. Dopo aver astratto e correlato i dati provenienti dai monitor, il modello crea una macchina virtuale, una rappresentazione del sistema che comprende tutti gli host presenti nella rete e la rete stessa.

I livelli sono sei, rappresentanti ognuno di essi il risultato di una trasformazione effettuata sui dati:

1. il primo livello rappresenta i dati dei vari log forniti dal sistema operativo e dal LAN monitor;

2. il secondo livello è quello degli *eventi*, cioè gli eventi "notevoli" sugli host e sulla rete;

3. al terzo livello vengono creati i *soggetti*. Ad esempio, sulla rete un utente viene identificato tramite un NID, per cui ogni sessione creata dall'utente sulla rete avrà assegnato lo stesso NID;

4. il quarto livello utilizza le *informazioni contestuali*, sia temporali che spaziali. Ad esempio, comportamenti considerati normali in alcuni intervalli di tempo possono essere considerati sospetti in altri intervalli di tempo. Ad esempio, si possono eseguire attività di manutenzione del sistema solo durante la notte. Un altro esempio è dato da ripetuti tentativi di login falliti in sequenza che sono considerati come tentativi di intrusione.

5. questo livello considera le *minacce* per la rete e per gli host, tramite una combinazione tra gli eventi e le informazioni contestuali. Le minacce sono classificate in *abuse*, cambiamenti allo stato di protezione del sistema, *misuse*, comportamenti che violano la politica ma non cambiano lo stato di protezione della macchina, e *suspicious acts*, comportamenti che non violano la politica del sistema ma sono comunque degni di nota;

6. all'ultimo livello viene presentato un valore tra 1 e 100 che rappresenta lo stato di sicurezza della rete dove più alto il numero più insicura la rete. Questo valore è una funzione di tutte le minacce e di tutti i soggetti presenti sulla rete.

3.5.5 Un modello basato sull'analisi delle transizioni di stato

Il modello descritto in [54] parte dalla constatazione che nei sistemi di rilevamento delle intrusioni che si basano sulle firme, dato un certo scenario di penetrazione, possono esserci

molteplici variazioni della stessa penetrazione che producono differenti sequenze di *audit*, cioè eventi registrati su file di log relativi alla sicurezza del sistema. Di conseguenza, anche una piccola variazione di uno scenario di attacco può non essere rilevata come tentativo di intrusione.

In secondo luogo, si riconosce che uno dei limiti dei sistemi di rilevamento delle intrusioni è quello di non essere in grado di contenere i danni prodotti da un attacco o di prevedere, con un certo grado di confidenza, un tentativo di intrusione per avvertire in anticipo il responsabile della sicurezza del sistema in modo che siano adottate contromisure che impediscano che il sistema sia compromesso.

Infine, si nota che le regole presenti nella base di dati difficilmente vengono aggiornate o raramente ne vengono create nuove.

Nell'articolo viene proposta una metodologia per rappresentare una penetrazione usando un *diagramma di transizione di stati*; per fare questo, si parte della premessa che tutte le penetrazioni:

a) richiedono che l'attaccante sia in possesso di alcuni prerequisiti di accesso al sistema da penetrare, quali, ad esempio, accesso a certi file, o possesso d'informazioni relative alla sicurezza di una componente del sistema;

b) permettono all'attaccante di ottenere certi privilegi che prima non possedeva. Ad esempio, accesso non autorizzato a certi file, accesso ai privilegi di un altro utente.

Quindi, tutte le sequenze di azioni eseguite da un attaccante possono essere viste come transizioni da uno *stato iniziale* ad uno *stato finale* in cui il sistema è compromesso, dove uno stato è tutto ciò che rappresenta il sistema, ad esempio la memoria e i registri del processore.

Vengono definiti i seguenti stati:

- lo *stato iniziale*: lo stato del sistema prima della penetrazione;
- lo *stato compromesso*: lo stato del sistema dopo che è avvenuta la penetrazione con successo;
- gli *stati di transizione*: gli stati intermedi, in cui l'attaccante esegue le azioni che portano a penetrare nel sistema con successo.

Quelle azioni che sono necessarie per il successo dell'attacco sono dette le *signature actions* (firme). L'analista del sistema deve quindi essere in grado di identificare il minimo numero di firme che compongono una penetrazione e usare tali firme per creare dei diagrammi di transizione di stato. Questi diagrammi permettono di individurare in maniera precisa quali sono i requisiti necessari per una penetrazione in un sistema.

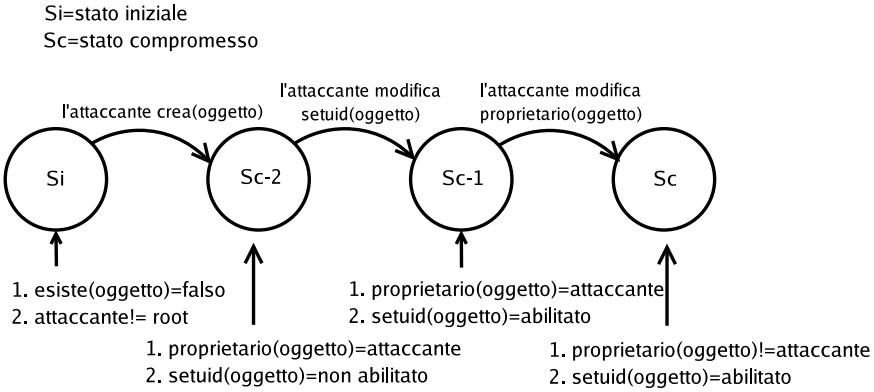


Figura 3.4: Esempio di diagramma di transizione di stati

Consideriamo, ad esempio, il seguente scenario di penetrazione. Una serie di comandi porta all'acquisizione da parte dell'attaccante dei privilegi di root:

```
user%cp /bin/csh /usr/spool/mail/root
user%chmod 4755 /usr/spool/mail/root
user%touch x
user%mail root < x
user%/user/spool/mail/root
root#
```

Più in dettaglio, l'attaccante sfrutta una vulnerabilità nel programma *mail*, tale per cui non viene resettato il bit setuid del file su cui vengono accodati i messaggi da inviare e viene cambiato il proprietario del file. La fig. 3.4 rappresenta il diagramma di transizione di stati per questo scenario.

L'analisi delle transizioni di stato viene usata come metodo per rappresentare le sequenze di azioni che un attaccante deve eseguire per effettuare una penetrazione con successo, e ha come obiettivo quello di identificare tutti gli scenari di attacco noti che conducono ad uno stato finale in cui il sistema è compromesso.

I log del sistema identificano i cambiamenti di stato effettuati sugli attributi del sistema monitorato da parte degli utenti: questi cambiamenti di stato sono confrontati con i diagrammi di transizione di attacchi noti, alla ricerca di eventuali intrusioni.

Lo strumento creato, che implementa queste tecniche, è chiamato *State Transition Analysis Tool* (STAT) e ha appunto il compito di analizzare i file di log del sistema ed estrarre informazioni di cambiamento di stato che vengono comparate con le rappresentazioni delle intrusioni presenti nella base di dati delle firme.

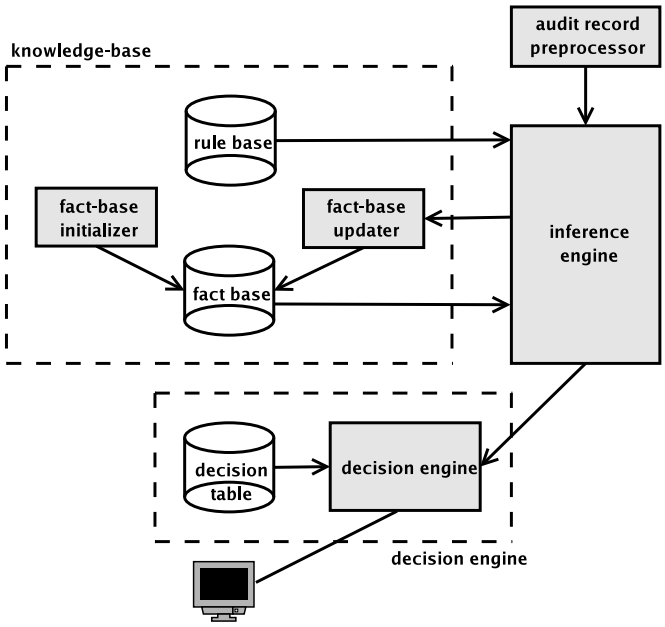


Figura 3.5: L'architettura centrale di STAT

La parte centrale di STAT, illustrata nella fig. 3.5, è formata dall'*inference engine*, dalla *knowledge-base* e dal *decision-engine*. I dati derivanti dai vari meccanismi di log sono inviati ad un *audit record preprocessor* che formatta i dati per essere passati all'inference engine. L'inference engine ha il compito di controllare i cambiamenti di stato del sistema, che vengono riscontrati analizzando i record di log, e di comparare questi cambiamenti di stato con gli scenari di penetrazione presenti nella knowledge-base. Il decision engine controlla i progressi che l'inference engine fa durante l'analisi delle transizioni di stato. Quando viene trovato un riscontro tra le transizioni di stato del sistema e uno scenario presente nella knowledge-base, il decision engine esegue un'azione in risposta a quest'evento.

3.5.5.1 L'audit record preprocessor

Questa componente ha il compito di ricevere i dati di log del sistema, filtarli e formattarli: la struttura dei record dopo essere stati formattati corrisponde ad una "signature action" che viene usata per rappresentare le transizioni di stato nei diagrammi. Il formato dei record dopo la formattazione è il seguente:

```
<Subject_ID, Subject Permissions, Actions, Object ID,
 Object Owner, Object Permissions>
```

Esempi di azioni eseguite su un file sono: creare, leggere, scrivere, modificare, eseguire o modificare gli attributi.

3.5.5.2 La knowledge-base

La knowledge-base è formata da due componenti: la *fact-base*, contenente i fatti, che sono asserzioni riguardanti lo stato corrente del sistema, e la *rule-base*, contenente le regole di inferenza usate dal sistema per inferire nuovi fatti a partire dai fatti correnti combinati con i nuovi dati ricevuti.

Tutte le regole contenute nella rule-base che identificano la firma di un attacco, vengono complessivamente classificate come una *rule chain*.

Le regole sono composte da tre campi:

- descrizione dello stato, contenente asserzioni sullo stato all'interno di un diagramma;
- la "signature action", che identifica l'azione che porta lo stato del sistema dallo stato corrente, presente nel precedente campo, allo stato successivo;
- la dipendenza delle regole. Questo campo impone un ordinamento tra gli stati e le azioni poiché identifica tutte le altre regole nella "chain" che devono essere soddisfatte prima di poter effettuare una transizione di stato.

3.5.5.3 L'inference engine

Questa componente contiene gli algoritmi utilizzati per monitorare e confrontare i cambiamenti di stato che avvengono nel sistema monitorato con la rappresentazione delle transizioni di stato contenute nella knowledge-base.

3.5.5.4 Il decision engine

Il decision engine ha il compito di decidere che tipo di azioni devono essere prese in risposta ad un evento rilevato da parte dell'inference engine. Ad esempio, può avvisare il responsabile della sicurezza oppure bloccare l'esecuzione della successiva "signature action" in modo da prevenire il completamento di un'intrusione, evitando che tutti i passi che formano una "chain" vengano eseguiti.

3.5.6 Monitoraggio delle chiamate di sistema

In [33] viene descritto il sistema *Widsard*, un IDS in grado di rilevare attacchi ad un host analizzando le sequenze delle chiamate al sistema operativo eseguite dai processi in esecuzione sull'host.

Viene specificato un formalismo per descrivere le anomalie da rilevare; questo formalismo estende quello delle espressioni regolari e permette di specificare il comportamento dei processi da monitorare in due passi:

a) vengono definiti tutti gli eventi interessanti;

b) il comportamento degli eventi interessanti viene descritto usando una *behavior expression*, definita componendo gli eventi interessanti tramite degli operatori, e associando alle occorrenze degli eventi delle condizioni su un insieme di variabili globali (il cui valore appartiene ad un dominio finito).

Quattro sono i parametri utilizzati per la definizione di un evento:

- la chiamata di sistema *Ca*;
- i parametri della chiamata *params*;
- una condizione *cond*;
- una sequenza di istruzioni *SI*.

Per cui un evento viene definito come:

```
Ca(params)(Cond)<SI>
```

dove i parametri "Cond" e "SI" sono opzionali. "Cond" è espressa in termini di parametri della chiamata e di variabili globali. Una chiamata di sistema a "Ca" riscontra l'evento solo se "Cond" è verificata: in caso affermativo, le istruzioni "SI" sono eseguite per aggiornare il valore delle variabili globali.

Ad esempio, tutte le chiamate a `open` per aprire il file `/etc/passwd`, sono riscontrate dall'evento:

```
open(/etc/passwd, -)
```

In maniera analoga, lo stesso evento può essere descritto come:

```
open($V, -)($V ==/etc/passwd)
```

dove viene introdotta una variabile "V" sulla quale viene definita una condizione.

Sono anche definiti i seguenti operatori, che possono essere utilizzati per comporre espressioni (dove un'espressione descrive un comportamento):

- composizione sequenziale: $B_1;B_2$ identifica un comportamento descritto prima da B_1 e poi da B_2; ad esempio, l'espressione:

  ```
  open(/etc/passwd,-)=fd; close(fd,-)
  ```

  definisce un comportamento di un processo che, prima apre il file delle password, e poi lo chiude;

- composizione alternativa: $B_1|B_2$ identifica un comportamento che è uguale a quello espresso da B_1 o da B_2;

- iterazione: B^* identifica un comportamento che itera zero o più volte il comportamento descritto da B.

Per monitorare un processo, viene introdotto *S(MP)* che è l'"ombra" di *MP*, il processo monitorato. Il comportamento di "S(MP)" può essere descritto in termini di due interpreti: il *behavior recognizer* e l'*evaluator*. Il primo ha come input un insieme di espressioni che definiscono i comportamenti anomali da monitorare: l'interprete deve riconoscere questi comportamenti analizzando le chiamate di sistema eseguite dal processo monitorato. L'"evaluator", invece, ha come input le condizioni che devono essere valutate prima dell'invocazione della chiamata di sistema, e le istruzioni che devono essere eseguite dopo la chiamata di sistema. È compito del "behavior recognizer" quello di attivare l'"evaluator" ogni volta che ciò è richiesto da una delle "behavior expression".

3.5.7 LIDS

LIDS (*Linux Intrusion Detection System*) [13] è un insieme di patch che vengono inserite nel kernel di Linux per implementare direttamente nel kernel il modello di sicurezza *reference model* e il modello *Mandatory Access Control*.

I problemi che LIDS cerca di risolvere sono [52, 53]:

- il file system di Linux non è protetto: ad esempio, il file `/bin/login` può essere modificato da un attaccante per installare un *trojan horse*. Questo file, come molti altri, viene modificato raramente, ad esempio al momento di eseguire un aggiornamento del sistema, e quindi dovrebbe essere protetto;

- i processi sotto Linux non sono protetti: se l'attaccante ottiene i privilegi di root, può terminare qualsiasi processo tramite la `kill`;
- l'amministrazione del sistema non è protetta. Anche in questo caso, se l'attaccante ottiene i privilegi di root, può caricare moduli nel kernel, modificare regole del firewall.

LIDS ha quindi il compito di:

- proteggere i file importanti;
- proteggere i processi imporanti;
- impedire modifiche al kernel. Ad esempio, impedire di inserire moduli nel kernel.

Per proteggere il file system, LIDS suddivide i file e le directory in queste categorie:

- *Read-Only*: nessuno può modificare questi file e directory;
- *Append-Only*: i file e le directory che appartengono a questa categoria possono solamente crescere in dimensione;
- *Exception*: i file e le directory di questa categoria non sono protetti.

Inoltre, è possibile specificare che nessuno, compreso root, possa montare o smontare il file system. LIDS protegge il sistema anche dalle operazioni di *RAW I/O* verso il disco, ad esempio verso il *master boot record*.

Per quanto riguarda la protezione dei processi importanti, LIDS permette di specificare quali sono quei processi che non possono essere terminati. Invece, per proteggere il sistema, prima di avviare il sistema deve essere effettuta l'operazione di *sealing* del kernel, che fa sì che quando il sistema sarà avviato, non sarà più possibile effettuare modifiche al kernel.

LIDS fornisce anche un rilevatore di *port scan* che riesce a rilevare gli *half-open scan* e anche gli *stealth scan* (ad esempio, FIN, Xmas, Null) direttamente nel kernel.

Quando LIDS rileva un tentativo di intrusione:

- emette un dettagliato messaggio relativo alla violazione avvenuta in un file di log del kernel: questo file è protetto da LIDS (anche per quanto riguarda attacchi di *flooding* al file di log);
- può inviare messaggi tramite e-mail all'amministratore del sistema;
- termina la sessione dell'utente che sta violando le politiche di sicurezza del sistema.

Ai processi vengono assegnate le *capability*, che identificano cosa un processo può fare. Esse sono definite nel kernel tramite una maschera di 32 bit.

Di seguito, presentiamo una lista di capability che possono essere abilitate a livello di sistema:

- *CAP_LINUX_IMMUTABLE*: protegge dalla scrittura i file e le directory che sono etichettati come immutabili;
- *CAP_NET_ADMIN*: previene la configurazione della rete (sia delle interfacce, che dei file di configurazione: regole del firewall, etc);
- *CAP_SYS_MODULE*: disabilita la capacità di inserire e rimuovere moduli dal kernel;
- *CAP_SYS_RAWIO*: impedisce il "RAW I/O";
- *CAP_SYS_ADMIN*: disabilita molte funzionalità di amministrazione del sistema.

Grazie a questo strumento, Linux viene protetto dall'abuso dei privilegi di cui dispone l'utente root e dal fatto che un attaccante che riesca ad ottenere l'accesso al sistema come root può controllare tutto il sistema.

Capitolo 4

Psyco-Virt

Contenuto

In questo capitolo viene descritta la progettazione e la realizzazione di un sistema distribuito per il rilevamento delle intrusioni, chiamato *Psyco-Virt*, che fa uso di tecniche di *introspezione*.

4.1 Virtual Machine Introspection

La scelta tra Host IDS e Network IDS è dettata anche da questi fattori, oltre a quelli descritti nel cap.3:

- l'Host IDS ha una visione molto accurata di quanto avviene sull'host, poiché è eseguito sulla stessa macchina che deve proteggere, ma d'altro canto è soggetto agli stessi attacchi di cui è soggetto l'host su cui è installato. Quindi, una volta compromesso l'host, tutto il software in esecuzione su di esso, tra cui proprio l'IDS, non è più affidabile;
- il Network IDS è più resistente agli attacchi che vengono effettuati agli host monitorati, poiché è installato su una macchina a sé stante, solitamente configurata per essere meglio difesa. Esso non può però avere la visione completa che ha un Host IDS sulle azioni che avvengono all'interno degli host, come, ad esempio, il monitoraggio delle chiamate di sistema, ed è quindi più facile eluderne il controllo.

In generale, due sono i modi per attaccare un IDS:

a) **attacco diretto**: vengono modificati i componenti che formano un IDS mediante manomissione diretta, così che un IDS non riconosce o non registra più le attività sospette;

b) **evasione**: vengono eseguite delle attività create ad arte, in maniera tale che l'IDS sia ingannato e non le riconosca come tentativi di intrusione. L'evasione è quindi il processo tramite cui viene modificato il formato o la temporizzazione di un attacco per cui l'attacco non viene più rilevato come tale, ma gli effetti dell'attacco rimangono immutati: l'attaccante, in questa maniera, spera di evitare che gli IDS riconoscano l'attacco [59].

La differenza tra queste due metodologie, attacco diretto ed evasione, sta nel fatto che nel primo caso si cerca di bloccare l'azione dell'IDS; nel secondo caso, invece, si cerca di mascherare come "innocue" le attività di intromissione. In quest'ultimo caso, si può cercare di aumentare la visibilità che l'IDS ha del sistema monitorato in modo che siano analizzati più eventi e più componenti del sistema, e che diventi così più difficile ingannare l'IDS. In questa maniera, però, dato che per ottenere una visibilità maggiore l'IDS deve essere a stretto contatto con gli host da monitorare, i benefici che si ottengono sono controbilanciati da un più debole isolamento tra l'IDS e gli attacchi che vengono eseguiti sugli host che sta monitorando.

Per cercare di soddisfare questi due requisiti apparentemente contrastanti, visibilità degli eventi e resistenza agli attacchi, una tecnologia particolarmente interessante è quella della *Virtual Machine Introspection* (VMI) [44]: l'idea centrale di questa tecnologia è quella di utilizzare la virtualizzazione per spostare l'IDS fuori dall'host che deve controllare e installarlo all'interno del Virtual Machine Monitor, per sfruttare la possibilità offerta dal VMM di ispezionare in maniera diretta lo stato della macchina virtuale che esegue l'host che si vuole monitorare. Si cercano così di ottenere sia i benefici propri di un Host IDS, la visibilità dell'host, che quelli di un Network IDS, la maggiore resistenza agli attacchi (vedi fig. 4.1).

Utilizzando questa metodologia, si possono quindi ottenere i seguenti vantaggi:

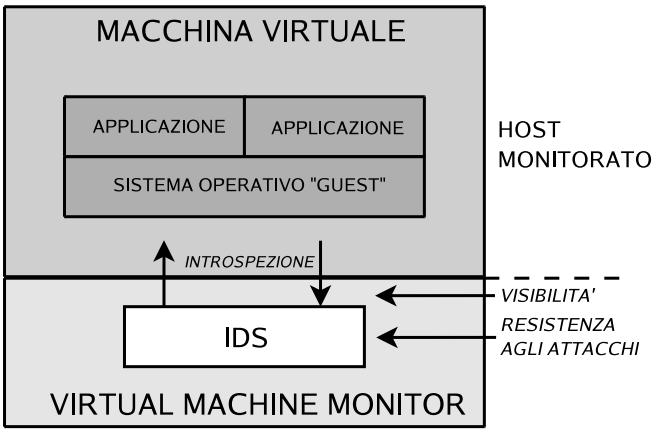


Figura 4.1: Introspezione

- maggior visibilità dell'host e capacità di introspezione dello stato delle macchine virtuali;
- maggiore resistenza agli attacchi, dovuta al fatto che l'IDS risiede su un dominio diverso rispetto a quello dell'host. Infatti, l'IDS non è installato sull'host, ma "sotto" di esso, nel VMM.

Un IDS realizzato usando questa tecnologia può osservare in maniera diretta lo stato dell'hardware della macchina virtuale, come i registri del processore, i buffer allocati, lo stato dei registri delle periferiche. Grazie ai dati osservati, l'IDS può inferire informazioni sullo stato dei vari livelli software della macchina virtuale osservata, per rilevare eventuali attacchi.

Inoltre, dato che un IDS tradizionale si affida ai controlli di sicurezza offerti da un sistema operativo per garantire la protezione e l'isolamento tra le applicazioni e l'IDS stesso, solitamente l'IDS è suscettibile di manomissioni se un host viene compromesso, perché una volta che il sistema operativo è stato compromesso, tali controlli possono essere facilmente disabilitati. Invece, un IDS realizzato tramite la Virtual Machine Introspection, anche in caso di compromissione del sistema operativo dell'host, può ugualmente osservare lo stato della macchina virtuale su cui è in esecuzione il sistema operativo compromesso ed effettuare azioni di ripristino.

Un altro vantaggio è conseguenza del fatto che in un'architettura Host IDS se l'IDS ha dei malfunzionamenti, e deve essere riavviato, è molto probabile che tutto il sistema debba essere riavviato, come avviene se l'IDS risiede nel kernel. Invece, nel caso di un IDS che sfrutti la tecnologia VMI, per poter cominciare nuovamente le attività di monitoraggio si deve semplicemente sospendere la macchina virtuale da monitorare solo per il periodo in cui si riavvia l'IDS che era stato interrotto per qualche ragione.

Inoltre, nei casi in cui alcuni dei controlli di sicurezza messi in atto dal sistema operativo siano stati disabilitati, l'IDS con la tecnologia VMI può effettuare tali controlli di protezione

a sua volta per conto del sistema operativo (ad esempio, controllando che le interfacce di rete dell'host non siano messe in modalità promiscua).

Il punto di partenza concettuale per utilizzare con fiducia questa tecnologia, è l'assunto che un VMM sia più difficile da attaccare e da compromettere che un host "normale". Questa assunzione si basa sulle seguenti osservazioni:

- l'interfaccia verso un VMM è molto più semplice di quella verso un sistema operativo, e molte operazioni sono eseguite in modalità non privilegiata, senza quindi richiedere l'intervento del VMM;
- il numero molto minore di istruzioni di un VMM rispetto ad un sistema operativo tradizionale, e quindi la sua minor complessità, fanno sì che l'architettura risultante sia più semplice e sia più facilmente controllabile e verificabile la correttezza della sua implementazione.

Riguardo l'ultimo aspetto, nell'ambito del progetto Xen ultimamente molte attività sono focalizzate a spostare molte delle operazioni che avvengono nel dominio privilegiato in domini utente non privilegiati. Ad esempio, per associare ad un dominio non privilegiato il controllo del disco, ad un altro il controllo della scheda di rete e così via [48]. Questo processo di spostamento di funzionalità è simile a quello che avviene nei sistemi operativi a microkernel rispetto a quelli monolitici [49].

4.1.1 Controllo del sistema e livelli di attacco e difesa

Per capire come negli anni si siano evoluti i sistemi di attacco ad un sistema informatico e di pari passo i meccanismi in grado di rilevarli, prendiamo come esempio i *rootkit* [76], il cui scopo è quello di nascondere le tracce che un attaccante ha lasciato sul sistema per permettergli di ritornare successivamente senza essere rilevato.

Per cercare di ottenere il controllo del sistema, i rootkit hanno sempre cercato di installarsi al livello più basso possibile. Infatti, si è passati dai rootkit che rimpiazzano programmi *user-level* che, come nel caso di `ps` modificato, forniscono informazioni errate all'utente sui processi in esecuzione, ai rootkit *kernel-level* che nascondono dati sul sistema, modificando strutture dati del kernel [9]. Analogamente, alcuni sistemi di IDS vengono installati direttamente nel kernel, per identificare tentativi di modifiche alle strutture dati del kernel.

Di conseguenza, chi riesce ad occupare il livello più basso del sistema può controllare i livelli più alti, grazie a due meccanismi:

a) l'*astrazione*, che fornisce una visione delle risorse *ad hoc*;

b) l'*interposizione*, grazie a cui per ogni richiesta di accesso alle risorse, il livello più basso controlla ed esegue l'operazione richiesta.

Se il livello più basso è occupato dall'IDS, questo è in grado di rilevare i tentativi di intrusione e di bloccarli; se, invece, il livello più basso è occupato dell'attaccante, allora vengono utilizzate tecniche di evasione per nascondere la propria presenza ai sistemi di rilevamento delle intrusioni.

Visto che sia i sistemi di rilevamento delle intrusioni che i software utilizzati dagli attaccanti, come i rootkit o i malware, possono lavorare al livello più privilegiato, nella modalità kernel, nessuno dei due ha un effettivo "vantaggio" sull'altro per ottenere il controllo del sistema. Di conseguenza, possiamo utilizzare la virtualizzazione per installare strumenti di controllo ad un livello più basso di quello a cui lavorano gli attaccanti.

In questo modo, poiché i sistemi operativi vengono eseguiti da una macchina virtuale, è possibile controllare lo stato di ogni macchina virtuale, tramite tecniche di introspezione, alla ricerca di comportamenti sospetti che indichino tentativi di modifiche al kernel.

Così facendo, abbiamo il vantaggio di agire ad un livello più basso di quello che può sperare di ottenere un attaccante nel momento in cui installa un rootkit a livello kernel.

4.2 Architettura

Psyco-Virt è il prototipo che è stato realizzato per implementare i controlli di rilevamento delle intrusioni su macchine virtuali tramite tecniche di introspezione. L'architettura è distribuita e prevede una macchina virtuale privilegiata, che quindi ha accesso all'interfaccia di controllo esportata dal Virtual Machine Monitor, che svolge due compiti: a) esamina lo stato delle macchine virtuali tramite introspezione; b) riceve ed analizza tutti i messaggi relativi a tentativi di intrusione inviati, su una rete di controllo, dalle componenti del sistema installate sulle macchine virtuali. In caso di intrusioni o attacchi ad un host, la macchina virtuale privilegiata esegue delle azioni in risposta a tali eventi, agendo sullo stato di esecuzione delle macchine virtuali attaccate, ad esempio bloccandone l'esecuzione.

La macchina virtuale privilegiata può esaminare lo stato di ogni macchina virtuale: questo compito è facilitato dalla libreria di introspezione che permette sia di avere una visione di "basso livello", a livello di memoria e di registri, che una visione di "alto livello", che fornisce una visione delle strutture dati del kernel dei sistemi operativi in esecuzione sulle macchine virtuali.

Sulla macchina virtuale privilegiata sono presenti le componenenti che sono in comunicazione con le componenti del sistema installate sulle macchine virtuali monitorate. Infatti, l'architettura distribuita prevede di utilizzare anche agenti installati sulle macchine virtuali da monitorare che, come gli Host IDS, rilevano le intrusioni e gli attacchi verso gli host monitorati.

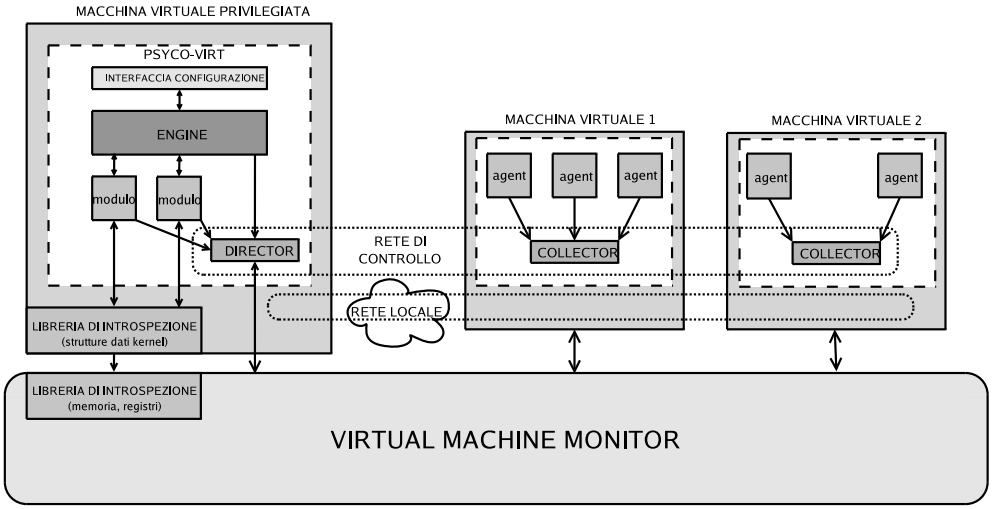


Figura 4.2: Architettura di Psyco-Virt

Questa architettura sfrutta quindi sia le tecniche di introspezione, per esaminare a livello del VMM lo stato di esecuzione delle macchine virtuali, che gli strumenti di rilevamento delle intrusioni, tramite cui gli agenti installati sulle macchine virtuali rilevano eventuali intrusioni/attacchi, segnalandole alla componente centrale del sistema.

Inoltre, le tecniche di introspezione vengono utilizzate anche per monitorare le stesse componenti del sistema installate sulle macchine virtuali, gli agenti, per rilevare eventuali compromissioni: in questo modo, anche in caso di attacchi agli host in esecuzione sulle macchine virtuali, si possono rilevare i tentativi di modifiche effettuati dagli attaccanti agli agenti per fornire dati alterati sullo stato dell'host.

L'architettura del sistema Psyco-Virt è presentata in fig. 4.2. Le parti fondamentali del sistema sono:

- la libreria di introspezione;
- l'engine;
- il director;
- i moduli;
- i collector;
- gli agent;
- l'interfaccia di configurazione;

- la rete di controllo.

Come si vede, è prevista una macchina virtuale privilegiata che ha accesso all'interfaccia di controllo del VMM per poter creare e distruggere le macchine virtuali e per accedere, tramite la *libreria di introspezione*, allo stato delle macchine virtuali. Questa macchina esegue l'*engine*, il *director* e i *moduli*. Le macchine virtuali monitorate eseguono, invece, gli *agent* che sono coordinati da un *collector* per macchina. Lo scambio di informazioni tra i collector e i director avviene tramite una *rete di controllo*. L'*interfaccia di configurazione* permette all'amministratore di sistema di configurare e di attivare Psyco-Virt.

4.2.1 La libreria di introspezione

La libreria di introspezione fornisce una visione a più livelli dello stato delle macchine virtuali in esecuzione. La libreria di introspezione di *basso livello* permette di esaminare direttamente lo stato delle macchine virtuali mettendo a disposizione la visione e l'accesso alla memoria, ai registri del processore e ai dispositivi di I/O associati alle macchine virtuali. La libreria di introspezione di *alto livello*, invece, fornisce delle funzioni che interpretano i dati di basso livello sullo stato delle macchine virtuali per avere una visione a livello di sistema operativo.

Tramite queste funzioni è possibile vedere direttamente le strutture dati del kernel del sistema operativo in esecuzione su una data macchina virtuale. Per fare questo, la libreria di introspezione di alto livello deve conoscere come è strutturato il kernel in esecuzione su una certa macchina virtuale. Come conseguenza di ciò, la libreria sa come sono organizzate le strutture dati usate dal kernel, ad esempio la lista dei processi pronti, e dove queste strutture sono allocate in memoria. Quindi, a partire dai dati forniti dalla libreria di introspezione di basso livello, la libreria di alto livello è in grado di ricostruire le strutture dati utilizzate dal kernel.

La libreria di introspezione permette così al sistema di rilevamento delle intrusioni di lavorare a livello della semantica dei sistemi operativi e di eseguire quindi in maniera più semplice le attività di monitoraggio.

Ad esempio, Psyco-Virt può richiedere alla libreria di introspezione di alto livello, per una data macchina virtuale (vedi fig. 4.3):

- la lista dei processi in esecuzione;
- se l'interfaccia di rete è in modalità promiscua;
- le pagine contenenti le istruzioni associate ad un processo identificato da un `PID`;
- la lista dei file aperti;

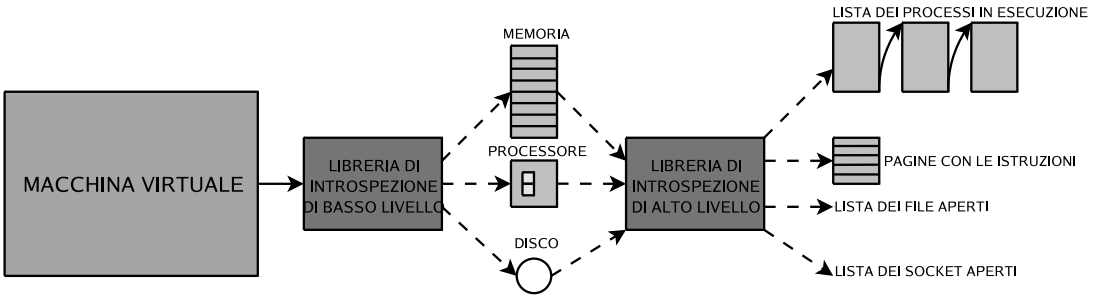


Figura 4.3: La libreria di introspezione

- la lista dei socket aperti.

La libreria di introspezione di basso livello, invece fornisce una visione a livello hardware dello stato della macchina virtuale. Ad esempio, è possibile richiedere alla libreria di basso livello una copia della memoria associata alla macchina virtuale monitorata. La memoria viene presentata come un array contiguo e, su questo array, Psyco-Virt può, ad esempio, effettuare dei controlli di *pattern-matching* di firme.

4.2.2 Engine

L'engine ha il compito di coordinare l'azione di tutte le componenti essenziali per il rilevamento delle intrusioni: il director, i moduli e i collector. Esso:

- interagisce direttamente con il director e con i moduli, attivandoli all'avvio del sistema;
- risponde alle intrusioni segnalate dai moduli e dagli agent, tramite i collector, eseguendo azioni specificate. Ad esempio, l'engine può bloccare l'esecuzione della macchina virtuale su cui gli agent o i moduli hanno rilevato dei tentativi di intrusione o inviare un messaggio di log alla console.

4.2.2.1 Le azioni

Le azioni sono eseguite ogni volta che un messaggio è ricevuto dal director e analizzato dall'engine: se all'evento segnalato da parte dei moduli o degli agent è associata un'azione, questa viene eseguita. Le componenti del sistema che segnalano i tentativi di intrusione specificano nel messaggio inviato al director il livello di gravità associato all'evento segnalato. In fase di configurazione del sistema vengono specificate le azioni che devono essere eseguite dall'engine in risposta agli eventi ricevuti, a seconda dei livelli di gravità associati.

Le azioni possono essere di due tipi: quelle che sono predefinite in Psyco-Virt oppure quelle che sono definite dagli utenti.

Azioni predefinite. Le azioni predefinite di Psyco-Virt possono:

- sospendere l'esecuzione di una macchina virtuale;
- bloccare l'esecuzione di una macchina virtuale e salvarne il contenuto della memoria su un file su disco;
- chiudere la connessione della macchina virtuale alla rete locale;
- uccidere un processo, tramite la `kill`;
- disconnettere un utente connesso ad una macchina virtuale;
- inviare un messaggio alla console del programma.

È anche possibile associare più di un'azione ad uno stesso evento, ad esempio se si vuole sospendere l'esecuzione di una macchina virtuale e allo stesso tempo inviare un messaggio alla console. Le azioni hanno lo scopo di evitare che un attacco abbia successo nonché di informare l'amministratore del sistema di eventuali intrusioni.

Azioni personalizzate. In aggiunta alle azioni di default, un utente può definire altre azioni personalizzate. Nel file di configurazione verrà specificata il nome dell'azione e il modulo su cui l'azione è definita. In questo caso, l'utente è libero di definire le azioni che ritiene più opportune e di associarle a dei livelli di gravità. In questa maniera, visto che l'utente conosce quali sono le applicazioni in esecuzione sugli host da monitorare e la tipologia della rete, è in grado di stabilire in maniera più precisa quali sono le contromisure da effettuare in risposta ad un'instrusione o ad un attacco.

Ad esempio, nel caso di attacchi rilevati sulla rete locale, se su quella rete è presente un firewall, una regola personalizzata potrebbe essere quella di aggiungere una regola al firewall per bloccare determinati IP o porte relativi alle connessioni utilizzate per effettuare gli attacchi.

4.2.3 Director

Il director è attivato dall'engine, e ha due compiti fondamentali:

- è il punto centrale di raccolta dei messaggi di log relativi a tentativi di intrusione inviati sia dai collector che risiedono sulle macchine virtuali che dai moduli in esecuzione sul dominio privilegiato;
- interagisce attivamente con gli agent sia su richiesta dell'amministratore sia, se risulta necessario, autonomamente. Ad esempio, può richiedere agli agent di inviargli più

informazioni di quelle che ha ricevuto per stabilire in maniera più precisa se siamo in presenza di un tentativo di intrusione.

4.2.4 Moduli

I moduli vengono attivati e controllati dall'engine che, durante il funzionamento del sistema, può terminare o caricare i moduli dinamicamente. I moduli applicano le tecniche di introspezione per controllare lo stato delle macchine virtuali in esecuzione alla ricerca di intrusioni. Ogni modulo controlla un aspetto specifico della macchina virtuale, ad esempio:

- se i processi in esecuzione sulla macchina virtuale, segnalati dagli agenti, sono gli stessi che vengono rilevati tramite introspezione. Se i due insiemi contengono elementi diversi, siamo in presenza di un attaccante che ha modificato alcuni programmi di sistema, ad esempio quelli per mostrare i processi in esecuzione, in modo da nascondere i processi che vuole eseguire senza che siano notati;

- se le pagine contenenti le istruzioni appartenenti ad alcuni processi critici che sono in esecuzione, ad esempio le pagine dei processi associati agli agent, al WebServer o al server DNS, sono state modificate rispetto ad una versione di queste considerata "sicura". Se quelle pagine, che in alcuni sistemi sono in sola lettura, sono state modificate, allora un attaccante ha modificato un processo in esecuzione per eseguire istruzioni da lui inserite. In questo modo, l'attaccante può inserire una *backdoor* o un *keylogger* all'interno di tali processi;

- se sono state effettuate modifiche alle strutture dati del kernel del sistema operativo in esecuzione nelle macchine virtuali: ad esempio, modifiche alla tabella delle interruzioni, tramite cui un attaccante può installare delle versioni modificate delle chiamate di sistema che verranno utilizzate ogni volta che una chiamata di sistema viene invocata, ottenendo così pieno controllo sull'esecuzione di tutti i programmi.

I moduli possono essere di due tipi:

a) moduli che controllano gli eventi periodicamente. Ogni modulo implementa un'interfaccia comune che specifica le azioni che il modulo deve effettuare e l'intervallo di attesa che deve passare tra un'attivazione e la successiva del modulo. È compito dell'engine attivare all'avvio di Psyco-Virt i moduli specificati nel file di configurazione. Ogni modulo esegue le azioni specificate, dopodiché si mette in stato di attesa fino a che, passato l'intervallo specificato, viene attivato nuovamente dall'engine e può ripetere le stesse azioni;

b) moduli che vengono attivati in risposta a certi eventi. Questi moduli registrano con l'engine il tipo di evento di cui devono essere avvertiti: ad esempio, un modulo può richiedere di essere svegliato ogni volta che l'interfaccia di rete di una macchina virtuale viene messa in modalità promiscua. Quando i moduli vengono svegliati, all'occorrenza dell'evento, eseguono un'azione, ad esempio inviano un messaggio al collector.

Anche in questo caso, come per le azioni, un utente può usare i moduli predefiniti o definire i propri moduli.

4.2.4.1 Moduli predefiniti

I moduli presenti in Psyco-Virt sono:

- il modulo *moduleps*: controlla che i processi segnalati da una macchina virtuale, e ottenuti dall'engine richiedendoli al collector, siano gli stessi ottenuti tramite la libreria di introspezione. Se ci sono differenze, siamo in presenza di un attaccante che ha modificato i programmi che, come `ps`, mostrano i processi in esecuzione per nascondere le proprie azioni;

- il modulo *modulehash*: controlla che gli hash calcolati sulle pagine appartenenti a processi specifici, ad esempio processi associati ad un WebServer, non siano modificati durante l'esecuzione della macchina virtuale. Le pagine controllate sono quelle contenenti le istruzioni, come quelle nella sezione TEXT del formato dei file ELF sotto Linux. Poiché tale sezione è protetta in sola lettura, ogni modifica delle pagine contenute in quella sezione può far parte di un tentativo dell'attaccante di inserire delle istruzioni per ottenere accesso illegale al sistema. Gli attacchi possono avvenire, ad esempio, tramite *code injection* o *buffer overflow*[1];

- il modulo *modulesignature*: verifica la presenza di determinate firme nella memoria associata ad una macchina virtuale, ad esempio firme appartenenti a rootkit specifici;

- il modulo *modulepromiscous*: controlla se l'interfaccia di rete di una macchina virtuale verso la rete locale è stata messa in *modalità promiscusa*, modalità che permetterebbe all'attaccante di intercettare tutto il traffico che arriva all'interfaccia di rete;

- il module *moduleidt*: controlla se il vettore delle interruzioni è stato modificato. Questa modifica permetterebbe all'attaccante, ad esempio, di installare delle chiamate di sistema modificate.

[1] un controllo simile sul kernel, effettuato tramite coprocessore, è descritto in [57].

Come già detto, i moduli sono attivati all'avvio dall'engine, secondo quanto specificato nel file di configurazione, oppure sono attivati dinamicamente su richiesta dell'amministratore del sistema.

4.2.5 Agenti

Gli agenti sono installati sulle macchine virtuali da monitorare e ognuno di essi controlla un diverso aspetto di tali macchine. Gli agenti possono essere *stand-alone*, se sono creati per interagire col sistema alla ricerca di compromissioni, oppure possono fare da *wrapper* verso altri strumenti per il rilevamento delle intrusioni. In questo caso, hanno il compito di convertire l'output prodotto dagli IDS nel formato dei messaggi usato per le comunicazioni col collector.

Ad esempio, sono stand-alone gli agenti che controllano:

- i tentativi di accesso senza successo, analizzando le informazioni che ottengono dai file di log del sistema;
- le risorse utilizzate dai processi, alla ricerca di abusi nell'uso delle risorse.

Nel caso di agenti "wrapper", è così possibile usare strumenti già esistenti, ad esempio chkrootkit [4] o Snort [22] e, in caso di una segnalazione relativa ad un'intrusione da parte di questi strumenti, l'agente provvede ad informare il collector dell'evento, convertendo in maniera adeguata il messaggio ricevuto dagli strumenti di rilevamento delle intrusioni, nel formato del messaggio utilizzato dal protocollo tra l'agent e il collector.

Gli agenti hanno una modalità di funzionamento simile a quella dei moduli, per cui vengono attivati dal collector all'avvio di Psyco-Virt, ed eseguono le loro azioni ad intervalli regolari. Inoltre, essi hanno le stesse caratteristiche degli Host IDS e, quindi, monitorano le chiamate di sistema, i file di log delle applicazioni critiche, il filesystem e i record di audit. In caso di eventi sospetti, inviano una segnalazione al collector.

4.2.5.1 Agenti predefiniti

Gli agenti presenti di default con Psyco-Virt sono:

- *agentlogin*: controlla il file contenente i tentativi falliti di accesso al sistema; dopo un certo numero di tentativi falliti in un intervallo di tempo l'agente avverte il collector. I parametri sono configurabili dall'utente;
- *agentchkrootkit*: implementa un "wrapper" agli script presenti nel tool chkrootkit [4], che rileva diversi tipi di rootkit presenti sul sistema;

- *agentsnort*: la macchina virtuale che ospita questo agente esegue il programma Snort [22] e ha funzionalità di Network IDS. È una macchina virtuale che esegue solo questa applicazione. In alternativa, è anche possibile eseguire Snort sul dominio 0 e associare ad esso un modulo appropriato. L'agent fa da "wrapper" a Snort, codificando i messaggi segnalati da Snort, sui tentativi di intrusione e attacco via rete, nel formato di log specifico di Psyco-Virt;

- *agentstepping*: questo agente implementa un algoritmo per rilevare gli *stepping stone* sulla rete locale [82]. Gli "stepping stone" sono host che sono stati compromessi da un attaccante e che si trovano nel cammino tra la macchina dell'attaccante e quella obiettivo dell'attacco. L'attaccante utilizza questi nodi in successione, effettuando una serie di login da uno "stepping stone" all'altro, creando così una catena di nodi compromessi, per lanciare infine l'attacco dall'ultimo nodo presente in questa catena al nodo bersaglio dell'attacco. Questa strategia permette all'attaccante di celare l'origine dell'attacco. Questo agent controlla la rete locale e, se rileva la presenza di uno o più "stepping stone", invia un alert al collector.

Anche gli agenti possono essere personalizzati, da parte degli utenti, alla stessa maniera dei moduli.

4.2.6 Collector

Il collector deve fare da centro di raccolta per le segnalazioni inviategli dagli agent che controlla. Esso può avvertire il director ogni volta che un agent gli invia una segnalazione, ad esempio se la segnalazione indica un attacco in corso, oppure può filtrare le segnalazioni ricevute e inviare un messaggio al collector dopo un numero configurabile di eventi. Questo avviene quando il collector riceve segnalazioni simili da più agent (segnalazioni che sono in correlazione tra di loro) per cui, invece di inviare lo stesso messaggio per ogni evento segnalato, ne viene inviato uno solo che contiene tutte le informazioni relative all'evento segnalato e agli host coinvolti.

Inoltre, il collector può, su richiesta del director, inviare informazioni al director. Ad esempio, può inviare:

- informazioni sul sistema su cui è installato, come la lista dei processi in esecuzione, la lista dei file aperti. In questo modo, il director può confrontare questi dati ricevuti dal collector con i dati ottenuti tramite introspezione alla ricerca di eventuali differenze, indicanti manomissione dell'host monitorato;

- ulteriori dati richiesti agli agent, nel caso in cui l'engine abbia bisogno di altri dati oltre a quelli già ricevuti. Ad esempio, nel caso di segnalazioni ricevute da vari collector in un breve intervallo di tempo, l'engine può richiedere, tramite director, agli agent ulteriori dati da analizzare e, confrontando questi dati, cercare di capire se siamo in presenza di un attacco che coinvolge diversi host.

Inoltre, il director può inviare ai collector il file di configurazione, qualora questo sia stato modificato. Esiste un collector per ogni macchina virtuale: questo collector controlla tutti gli agent installati sulla stessa macchina virtuale. I collector utilizzano la rete di controllo per inviare le segnalazioni al director.

4.2.7 La rete di controllo

Per scambiare informazioni tra i collector e il director, viene creata una rete di controllo. Questa rete non è raggiungibile da nessun host, e può essere utilizzata solo dalle macchine virtuali su cui è in esecuzione Psyco-Virt, e solamente dai componenti di tale applicazione. Per garantire questa proprietà, viene creato un *bridge virtuale* e delle interfacce di rete virtuali per ogni macchina virtuale: una interfaccia *dummy* per il dominio 0 e l'interfaccia *eth*1 per i domini utente. Questo bridge connette solo le macchine virtuali e non è raggiungibile dall'esterno.

Ovviamente, ogni dominio, se connesso alla rete locale, utilizzerà un'interfaccia verso tale rete, ad esempio l'interfaccia *eth*0, per ricevere e inviare pacchetti sulla rete locale. In fig. 4.4 è rappresentata la configurazione delle reti utilizzando Xen come VMM.

4.2.8 L'interfaccia di configurazione

L'interfaccia di configurazione è lo strumento tramite cui l'amministratore di sistema può avviare e sospendere Psyco-Virt e creare il file di configurazione contenente i parametri di esecuzione del sistema. L'interfaccia permette di specificare:

- i moduli da caricare: per ogni modulo, bisogna specificare il nome del file che contiene il modulo, e il nome della classe definita nel modulo;
- le macchine virtuali che devono essere monitorate: le macchine vengono specificate tramite il nome che è stato assegnato loro nel file di configurazione usato dal VMM per avviare le macchine virtuali, e l'indirizzo IP della loro interfaccia della rete di controllo e la porta su cui è in ascolto il collector;
- l'IP dell'interfaccia di controllo della macchina virtuale privilegiata e la porta di ascolto del director;

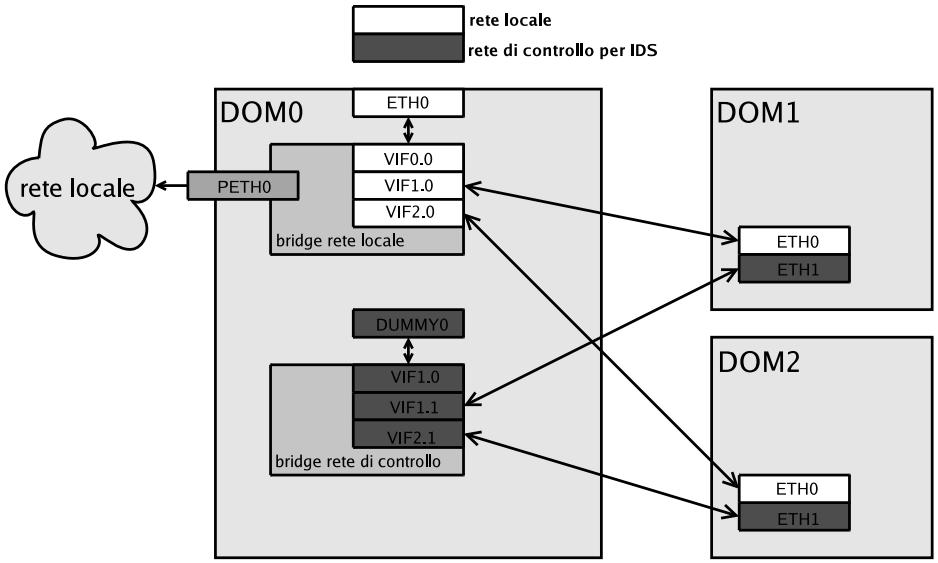


Figura 4.4: La rete di controllo su Xen per Psyco-Virt

- i livelli di logging: identificano la gravità associata ad un evento. Sono presenti dei livelli di default, ed è possibile rimuoverli e/o aggiungerne altri personalizzati;

- le azioni da eseguire in risposta agli eventi segnalati dai moduli o dai collector. Ad ogni livello di gravità (logging) definito viene associata un'azione che può essere predefinita o personalizzata. Se l'azione è personalizzata occorre specificare il nome della funzione da invocare e il file in cui è memorizzata questa funzione;

- il file di log utilizzato dall'engine per salvare i messaggi ricevuti dagli agent e dai moduli.

Oltre a queste funzionalità, l'interfaccia ha anche una console per ricevere informazioni sullo stato di Psyco-Virt e, nel caso in cui sia stato specificato, ricevere i messaggi di log inviati dagli agent e dai moduli e in risposta a certi eventi, ad esempio quando viene eseguita un'azione. Inoltre, tramite l'interfaccia è possibile sospendere l'esecuzione di una macchina virtuale o ripristinare l'esecuzione di una macchina precedentemente sospesa.

La configurazione viene salvata su un file in formato XML: questo file deve essere presente sia nella macchina virtuale privilegiata che su tutte le macchine virtuali monitorate.

Nella fig. 4.5 è illustrata l'interfaccia di configurazione di Psyco-Virt realizzata utilizzando la libreria TkInter [23]: questa interfaccia permette di effettuare tutte le operazioni descritte in precedenza. In particolare, nella parte destra dell'interfaccia è presente la console di log, dove sono mostrati i messaggi sullo stato di esecuzione di Psyco-Virt, ad esempio quali sono i moduli caricati, e i messaggi relativi alle azioni eseguite dall'engine in risposta a certi eventi o

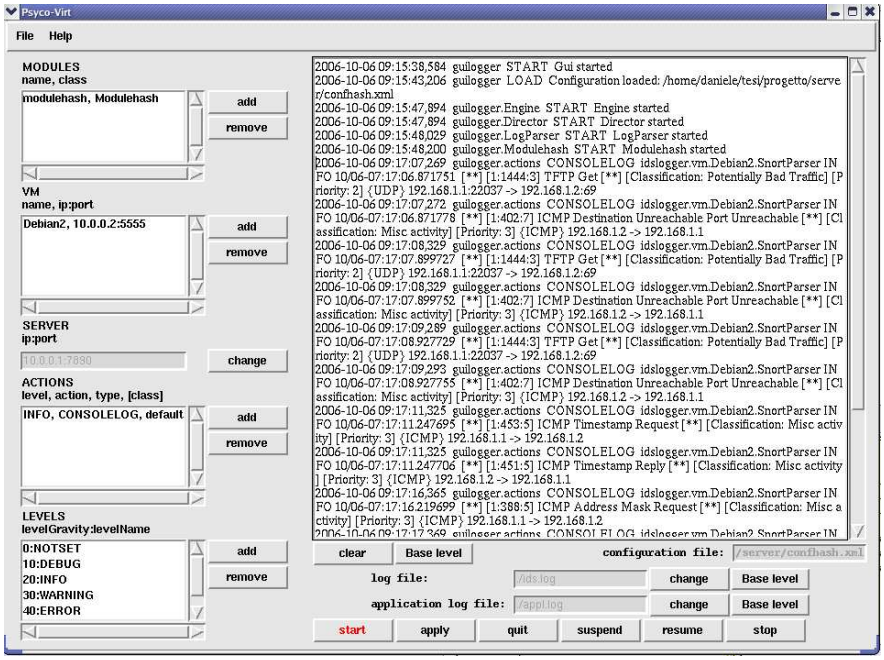


Figura 4.5: L'interfaccia di configurazione

gli alert inviati dai moduli o dagli agenti. Sempre sulla destra, in basso, sono presenti i comandi per avviare o terminare Psyco-Virt e per fermare o ripristinare l'esecuzione di una macchina virtuale. Sulla parte sinistra, invece, sono presenti i comandi che permettono di configurare i moduli da caricare, oppure di specificare quali sono le macchine virtuali da monitorare e gli agenti in esecuzione su di esse; inoltre, si possono settare i livelli di logging e le azioni associate ai livelli di logging; infine, è possibile specificare gli indirizzi IP delle macchine virtuali sulla rete di controllo.

4.3 Implementazione

Per l'implementazione di Psyco-Virt è stato scelto il linguaggio di programmazione Python [19]. Le ragioni di questa scelta sono:

- i tempi di sviluppo più rapidi rispetto ad altri linguaggi, come il C e il C++: ad esempio, in termini di numero di istruzioni, i programmi scritti in Python hanno una dimensione di circa un terzo/un quinto rispetto a quelli scritti in C, C++ [63];

- Python offre una libreria molto vasta: ad esempio, offre una libreria per il logging, una per le espressioni regolari e così via;

- Python ha una potenza espressiva elevata e offre strutture dati adatte per compiere operazioni di introspezione e ricerca, quali dizionari e liste;
- Python permette di includere e di integrare parti di codice scritte in C. Questa funzionalità è utile sia per il riutilizzo di funzioni presenti in librerie codificate in C, sia per aumentare l'efficienza di alcune parti di codice che richiedono prestazioni elevate.

Come Virtual Machine Monitor è stato scelto Xen (la versione 3.0.2-2); per il dominio 0 è stata utilizzata la distribuzione Fedora 5 di Linux [8]; per i sistemi operativi dei domini utenti, è stata utilizzata la distribuzione Debian di Linux [7].

4.3.1 Meccanismo di logging

Per scambiare messaggi tra i moduli, tra i collector e il director, viene utilizzata la libreria di logging fornita da Python. I moduli scrivono direttamente gli eventi su un file di log; per i messaggi ricevuti dai collector, invece, il director avvia sia un server in ascolto sulla rete di controllo, che riceve i messaggi dai collector e li scrive sul file di log, che un parser per leggere periodicamente il file di log.

I messaggi di log contengono quattro parametri:

1. **ora** e **data** dell'evento;
2. **nome**: viene utilizzato un meccanismo di naming gerarchico per individuare il componente del sistema che ha generato il log. Ad esempio:

   ```
   idslogger.vm.vmName.agentName
   ```

   identifica l'agente "agentName" in esecuzione sulla macchina virtuale denominata sotto Xen con l'identificativo "vmName";
3. **livello**: identifica il livello di gravità associato all'evento segnalato. I livelli di gravità di default sono mostrati nella tab. 4.1: più alto il numero, più alto è il livello di gravità associato all'evento;
4. **messaggio**: è il messaggio che l'agente o il modulo riportano per specificare il tipo di evento verificatosi più altre informazioni ritenute utili.

Durante la configurazione del sistema è possibile specificare il livello associato al meccanismo di logging, per indicare quali messaggi devono essere segnalati al director: più basso è il livello, più numerosi sono i messaggi inviati. Inoltre, sempre in fase di configurazione, si può

Nome livello	Gravità livello
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

Tabella 4.1: Livelli di gravità

associare un'azione ad ogni livello di gravità: l'azione viene eseguita ogni volta che il director riceve una segnalazione avente livello di gravità a cui è associato quella azione. Ad esempio, se al livello di gravità `INFO` è associata l'azione predefinita `CONSOLELOG`, ogni volta che il director riceve un alert avente come livello `INFO`, viene eseguita l'azione `CONSOLELOG`, che fa sì che il messaggio ricevuto dall'agent sia inviato alla console di Psyco-Virt.

Tutte le funzioni che definiscono le azioni personalizzate devono implementare la stessa interfaccia, che specifica come parametro della funzione una variabile *dizionario* contenente i quattro parametri del messaggio di log, identificati tramite le stringhe `TIME`, `NAME`, `LEVEL` e `MSG`.

Il file di log viene letto periodicamente dal parser che ha il compito di separare le quattro componenti del messaggio di log, di identificare il livello di gravità del messaggio e di eseguire l'azione associata all'evento. Il parser viene "svegliato" dal director o dai moduli ogni volta che è presente un nuovo messaggio nel file di log.

In maniera analoga, sulle macchine virtuali monitorate gli agenti scrivono i messaggi di log relativi a tentativi d'intrusione su un file di log diverso per ogni macchina virtuale. All'avvio il collector attiva un parser dei log, che viene svegliato dagli agenti ogni volta che nel file di log vengono accodati nuovi messaggi. In base alla configurazione, il parser può inviare il messaggio al director immediatamente dopo averlo parsato, oppure fondere in un unico messaggio un certo numero di messaggi. Il numero dei messaggi da fondere può essere assoluto o relativo ad un intervallo di tempo.

4.4 Test

I test hanno avuto lo scopo di verificare l'efficacia di Psyco-Virt, cioè la sua capacità di rilevare i tentativi di intrusione e attacco e di rispondere a questi tramite le azioni definite nel file di configurazione. Inoltre, i test hanno calcolato i tempi di overhead introdotti dall'utilizzo congiunto di Xen e di Psyco-Virt.

Per i test, come libreria di introspezione di alto livello è stata utilizzata la libreria XenAccess [29]. Poiché tale libreria è codificata in C, per utilizzarla è necessario importare da Python anche Ctypes [6], una libreria per Python che permette di richiamare codice scritto dal C all'interno del codice Python.

Inoltre, è stata utilizzata la libreria LibVirt [12] che consente di compiere operazioni sui domini utente, quali sospensione e salvataggio dello stato delle macchine virtuali, ed è codificata anche in Python.

XenAccess [29] è una libreria open source che permette di avere un'interfaccia semplificata verso i sistemi operativi in esecuzione sulle macchine virtuali, per poter così visionare lo stato delle macchine virtuali. Attualmente, la libreria XenAccess supporta solo il sistema operativo Linux.

Ad esempio, utilizzando la libreria XenAccess è possibile implementare funzioni che restituiscono:

- la lista dei processi in esecuzione su una determinata macchina virtuale;
- le pagine appartenenti ad un particolare processo;
- i moduli caricati dal sistema operativo in esecuzione su una macchina virtuale.

La libreria LibVirt, invece, consente di:

- ottenere informazioni dettagliate su un dominio: nome del dominio, ID del dominio, numero di *VCPU* (Virtual CPU) associate;
- sospendere e ripristinare l'esecuzione di un dominio;
- sospendere l'esecuzione di un dominio, e salvare il contenuto della memoria su disco.

Sono stati effettuati test nel caso in cui fosse presente un dominio utente e due domini utente. In fig. 4.6 è rappresentata l'architettura di Psyco-Virt utilizzata per i test nel caso in cui fossero presenti due domini utente.

Il sistema utilizzato per i test aveva le seguenti caratteristiche:

- processore: Intel Pentium con frequenza di clock 1.73 GHz;
- dimensione memoria fisica: 512MB;
- dimensione memoria virtuale associata ad ogni dominio utente: 64 MB;
- immagine kernel distribuzione Fedora: 2.6.17-1.2187_FC5;

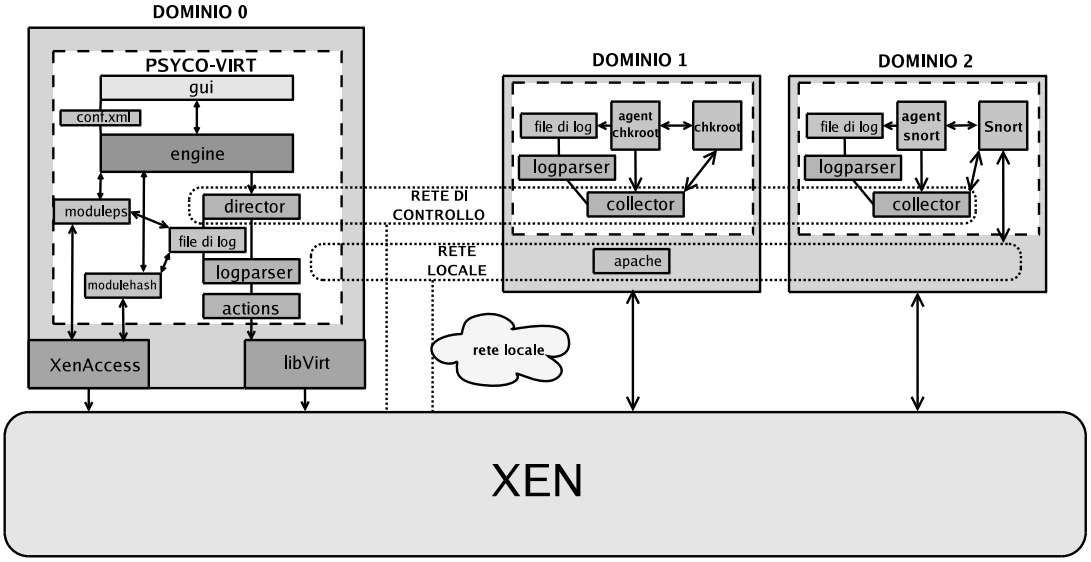


Figura 4.6: Architettura per i test

- immagine kernel dominio 0, distribuzione Fedora: 2.6.16-1.2133_FC5xen0;
- immagine kernel domini utente, distribuzione Debian: vmlinuz-2.6.16-xenU (compilata dai sorgenti);
- versione di Apache [2] utilizzata: 1.3.34;
- versione di Snort utilizzata: 2.3.3 (Build 14).

Di seguito viene mostrata la configurazione del file XML usata per i test. Questo file specifica che sul dominio 0 devono essere caricati i moduli *moduleps* e *modulehash*, sul dominio utente *Debian1* deve essere caricato l'agente *agentchkrootkit*, e sul dominio utente *Debian2*, invece, l'agente *agentsnort*. Inoltre, vengono specificati i livelli di gravità e le azioni da eseguire in risposta ad eventi segnalati con livello di gravità `INFO`, `WARNING` o `ERROR` e gli indirizzi IP delle interfacce della rete di controllo:

```
<configuration>
    <general>
        <levels>
            <level gravity="0" name="NOTSET">
            </level>
            <level gravity="10" name="DEBUG">
            </level>
            <level gravity="20" name="INFO">
            </level>
            <level gravity="30" name="WARNING">
```

```
            </level>
            <level gravity="40" name="ERROR">
            </level>
            <level gravity="50" name="CRITICAL">
            </level>
        </levels>
        <logging>
            <logFile type="ids" name="ids.log" level="0">
            </logFile>
            <logFile type="application" name="appl.log" level="100">
            </logFile>
            <logFile type="console" level="100">
            </logFile>
        </logging>
    </general>
    <domain0>
        <modules>
            <module name="modulehash" class="Modulehash">
            </module>
            <module name="moduleps" class="Moduleps">
            </module>
        </modules>
        <actions>
            <action execute="CONSOLELOG" type="default" level="INFO">
            </action>
            <action execute="STOP" type="default" level="WARNING">
            </action>
            <action execute="SAVE" type="default" level="ERROR">
            </action>
            <action execute="CONSOLELOG" type="default" level="ERROR">
            </action>
        </actions>
        <parameters>
        </parameters>
    </domain0>
    <domainU>
        <hosts>
            <vm ip="10.0.0.3" name="Debian1" port="5555">
               <agent name="agentchkrootkit" class="Agentchkrootkit">
                </agent>
            </vm>
            <vm ip="10.0.0.2" name="Debian2" port="5555">
                <agent name="agentsnort" class="Agentsnort">
                </agent>
            </vm>
            <server ip="10.0.0.1" port="7890">
            </server>
        </hosts>
    </domainU>
</configuration>
```

Il primo test è stato effettuato utilizzando il modulo *moduleps*, per confrontare la lista dei processi in esecuzione a due livelli: tramite introspezione e tramite il comando *ps* della distribuzione presente sulla macchina virtuale monitorata. Questo programma è stato sostituito con le versioni modificate disponibili nei rootkit `t0rn` [65], `lrk5`, `SUCKIT` e `tuxkit` che nascondono alcuni processi in esecuzione.

L'esempio seguente si riferisce al rootkit tuxkit, installato nella macchina virtuale monitorata. È stato modificato il file di configurazione del rootkit, il file `/dev/tux/.proc`, che descrive quali processi devono essere nascosti: aggiungendo la linea `0 1000`, vengono nascosti tutti i processi dell'utente 1000. Infatti, il modulo *moduleps*, tramite introspezione, ha ottenuto la seguente lista di processi:

```
1,2,3,4,5,6,7,8,9,16,37,38,40,39,556,586,988,1032,1038,
1052,1072,1075,1088,1089,1090,1091,1092,1093,1109,1150,
3090,3122,3191,3195,3196,3507,3520,3521,3522,3523,3524
```

Mentre, tramite richiesta al collector installato sulla macchina virtuale che esegue il comando `ps`, il modulo ha ottenuto la seguente lista:

```
1,2,3,4,5,6,7,8,9,16,37,38,40,39,556,586,988,1032,1038,
1052,1072,1075,1088,1089,1090,1091,1092,1093,1109,1150,
3090,3122,3191,3520,3521,3522,3523,3524
```

I due insiemi sono diversi; infatti, nell'ultimo caso, i processi nascosti che sono visibili tramite introspezione, sono:

```
1000   3195  sshd: daniele@pts
1000   3196  -bash
1000   3507  vi
```

Anche nel caso degli altri rootkit sono stati ottenuti risultati simili.

Il secondo test ha utilizzato l'agent *agentchkrootkit* che, come già detto, fa da wrapper per il tool *chkrootkit*. Di seguito mostriamo una parte dell'output di questo tool che mostra i file modificati da tuxkit nella macchina virtuale:

```
Checking 'du'... INFECTED
Checking 'find'... INFECTED
Checking 'ifconfig'... INFECTED
Checking 'killall'... INFECTED
Checking 'ls'... INFECTED
Checking 'netstat'... INFECTED
Checking 'ps'... INFECTED
```

```
Checking 'pstree'... INFECTED
Checking 'syslogd'... INFECTED
Checking 'tcpd'... INFECTED
Checking 'top'... INFECTED
Checking 'aliens'...
/dev/tux/tools/mirkforce/realnames /dev/tux/.proc /dev/tux/.addr
Searching for AjaKit rootkit default files and dirs...\
Possible AjaKit rootkit installed
Checking 'lkm'... You have     1 process hidden for ps command
chkproc: Warning: Possible LKM Trojan installed
```

L'agent effettua il parsing dell'output di chkrootkit, alla ricerca di stringhe indicanti compromissione dell'host, come quelle mostrate in precedenza, e avverte, tramite il collector, il director. Come conseguenza di tutto ciò, viene eseguita l'azione associata al livello di gravità segnalato.

Infine, sono stati effettuati i test usando l'agent *agentsnort*: questo agent effettua il parsing del file `/var/log/snort/alert` a tempi regolari (per il test, ogni secondo). Questo file contiene gli allarmi generati da Snort: in caso di nuovi alert presenti nel file, l'agent invia un messaggio al director. Per effettuare gli attacchi e/o per simularli sono stati utilizzati gli strumenti *Nessus* [15], *teardrop* e *Nmap* [17]. L'indirizzo 192.168.1.1 è stato assegnato alla macchina utilizzata per effettuare gli attacchi, il 192.168.1.2 è stato assegnato all'interfaccia per la rete locale della macchina virtuale su cui era in esecuzione Snort.

Il comando per eseguire Snort è stato:

```
snort -i eth0 -h 192.168.1.0/24 -A fast -N -c /etc/snort/snort.conf
```

Questi sono alcuni messaggi giunti alla console di Psyco-Virt che il director ha ricevuto dall'agentsnort, tramite il collector:

```
2006-09-28 09:52:07,551 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:06.671203 [**] [122:1:0] (portscan) TCP Portscan \
      [**] {PROTO255} 192.168.1.1 -> 192.168.1.2
2006-09-28 09:52:21,704 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:20.525196 [**] [121:4:1] Portscan detected from 192.168.1.1 \
      Talker(fixed: 30 sliding: 30) Scanner(fixed: 0 sliding: 0) [**]
2006-09-28 09:52:23,764 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:22.583800 [**] [1:1444:3] TFTP Get \
      [**] [Classification: Potentially Bad Traffic] [Priority: 2] \
      {UDP} 192.168.1.1:35848 -> 192.168.1.2:69
2006-09-28 09:52:23,765 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:22.583817 [**] [1:402:7] ICMP Destination Unreachable \
      Port Unreachable [**] [Classification: Misc activity] [Priority: 3] \
      {ICMP} 192.168.1.2 -> 192.168.1.1
2006-09-28 09:52:24,768 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:23.611745 [**] [1:1444:3] TFTP Get \
      [**] [Classification: Potentially Bad Traffic] [Priority: 2] \
```

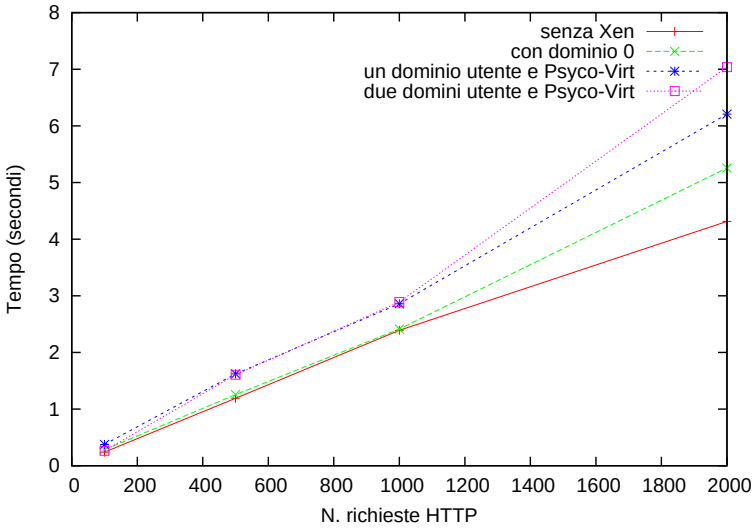
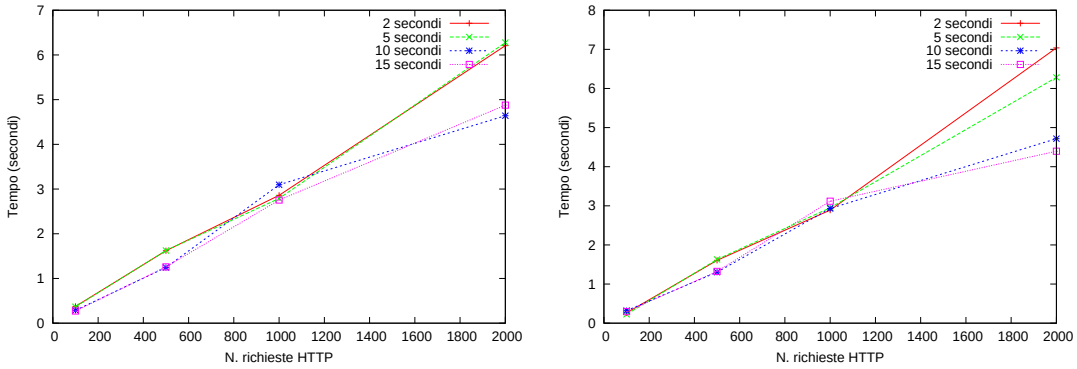


Figura 4.7: Differenze di tempi per le richieste HTTP

```
      {UDP} 192.168.1.1:35848 -> 192.168.1.2:69
2006-09-28 09:52:24,769 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:23.611770 [**] [1:402:7] ICMP Destination \
      Unreachable Port Unreachable [**] [Classification: Misc activity] [Priority: 3] \
      {ICMP} 192.168.1.2 -> 192.168.1.1
[...]
2006-09-28 09:52:53,810 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:52:52.431735 [**] [1:408:5] ICMP Echo Reply \
      [**] [Classification: Misc activity] [Priority: 3] \
      {ICMP} 192.168.1.2 -> 192.168.1.1
2006-09-28 09:54:48,385 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:54:47.237203 [**] [113:2:1] (spp_frag2) Teardrop attack \
      [**] {UDP} 192.168.1.1 -> 192.168.1.2
2006-09-28 09:55:17,774 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:55:17.232024 [**] [1:410:5] ICMP Fragment Reassembly \
      Time Exceeded [**] [Classification: Misc activity] [Priority: 3] \
      {ICMP} 192.168.1.2 -> 192.168.1.1
2006-09-28 09:55:28,833 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:55:27.994967 [**] [122:1:0] (portscan) TCP Portscan \
      [**] {PROTO255} 192.168.1.2 -> 192.168.1.1
2006-09-28 09:58:06,438 guilogger.actions CONSOLELOG idslogger.vm.Debian2.agentsnort \
      ERROR 09/28-07:58:05.069453 [**] [122:1:0] (portscan) TCP Portscan [**] \
      {PROTO255} 192.168.1.2 -> 192.168.1.1
```

Sono stati effettuati anche dei test per calcolare i tempi di overhead introdotti da Xen e Psyco-Virt con il modulo *modulehash*; sono stati calcolati i tempi di risposta a richieste HTTP ad un server Web Apache installato dapprima su un host senza che Xen fosse in esecuzione; successivamente, è stato avviato Xen e Apache è stato messo in esecuzione sul domino 0 (il



(a)

(b)

Figura 4.8: Tempi con un dominio utente (a) e due domini utente (b), con diversa frequenza di polling

dominio privilegiato); in seguito, è stata creata una macchina virtuale su cui era in esecuzione Apache ed è stato avviato anche Psyco-Virt con il modulo *modulehash* per controllare le pagine del processo associato al demone di Apache; infine, alla precedente configurazione è stata aggiunta un'altra macchina virtuale, sulla quale era in esecuzione Snort per effettuare controlli di intrusione sulla rete locale.

Sono state fatte 100, 500, 1000 e 2000 richieste al server Apache, e sono stati calcolati i tempi di risposta del server. I tempi di risposta, rappresentati in fig. 4.7, sono stati calcolati sulla media di 30 test identici. Nel passare da una configurazione nella quale Xen non era presente ad una in cui Xen era in esecuzione, i tempi hanno avuto un overhead del 2-20% circa; aggiungendo una macchina virtuale ed eseguendo Psyco-Virt, i tempi sono aumentati di circa il 20-30%; se, rispetto alla configurazione precedente, era anche in esecuzione un'altra macchina virtuale, i tempi di risposta hanno avuto un ulteriore overhead che, nel caso peggiore, è stato del 13%. I tempi riportati in figura, riferiti agli ultimi due casi, sono relativi al caso peggiore, quando cioè la frequenza di polling del modulo *modulehash* era la più alta (vedi test successivo).

Nella fig. 4.8(a) e 4.8(b) sono rappresentati i tempi calcolati effettuando gli stessi test del caso precedente, rispettivamente con un dominio utente e con due domini utente, al variare del tempo di polling del modulo *modulehash*, cioè il tempo tra due esecuzioni successive del modulo per il controllo delle pagine. Il test è consistito nel effettuare le richieste ad un server Web Apache installato su un dominio utente, e allo stesso tempo nell'eseguire il modulo per il controllo e per il calcolo dell'hash delle pagine contenenti le istruzioni del processo associato ad Apache ad intervalli di polling differenti (2, 5, 10 e 15 secondi tra una attivazione del modulo e la successiva).

Nel primo caso, con un dominio utente, passando dalla frequenza di polling più bassa a quella più alta, l'aumento complessivo dei tempi di risposta alle richieste è stato del 23%; nel

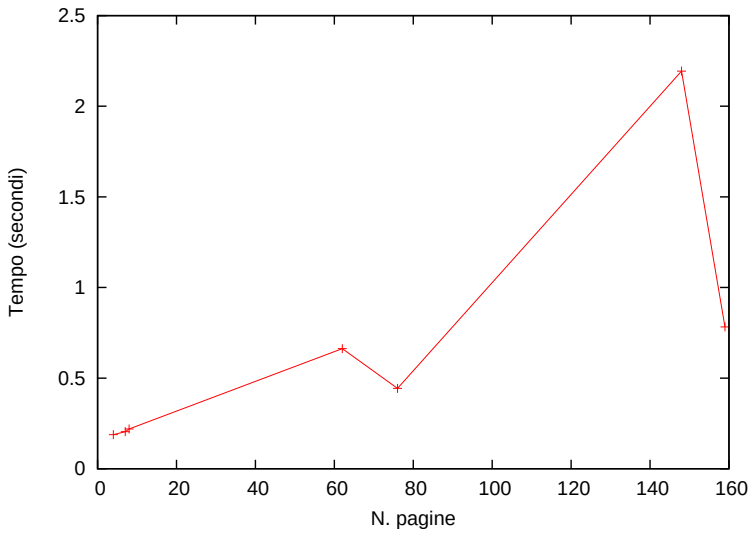


Figura 4.9: Tempi di copia delle pagine e calcolo dell'hash

secondo caso, con due domini utente, del 60%. Infatti, il modulo per la copia delle pagine e per il calcolo dell'hash sfrutta in maniera intensiva le risorse della macchina (il processore e la memoria in particolare), per cui se è attivato con una frequenza più alta i tempi di risposta alle richieste HTTP sono nettamente maggiori.

Nella fig. 4.9 sono rappresentati i tempi per ottenere una copia delle pagine contenenti le istruzioni di un processo in esecuzione su un dominio utente, tramite XenAccess, sommati a quelli per il calcolo dell'hash delle pagine stesse, tramite la funzione di hashing SHA512 presente nella libreria di Python. Il calcolo è stato effettuato compiendo le operazioni sulle pagine di 10 diversi processi in esecuzione su un dominio utente. Come si vede, benché ci sia una tendenza all'aumento dei tempi di copia e calcolo dell'hash in corrispondenza dell'aumento del numero delle pagine, alcuni valori sono variabili in quanto dipendenti dal *working set* dei processi: ciò fa sì che alcuni processi abbiano, in percentuale, molte delle loro pagine contenenti le istruzioni in memoria, mentre altri ne abbiano meno.

4.5 Psyco-Virt e lo stato dell'arte

Sono stati pubblicati già molti lavori che trattano l'utilizzo di tecnologie di virtualizzazione integrate ad architetture di sicurezza, tra cui ReVirt [41] che è un sistema di logging basato su macchine virtuali che ha scopi di *recovery*, *checkpoint* e *roll-back*: ReVirt è in grado di rieseguire, istruzione per istruzione, l'andamento di un sistema incapsulato dentro una macchina virtuale. SubVirt [60], invece, implementa una tecnica per installare un tipo di malware

denominato *Virtual-Machine Based Rootkit* (VMBR), tramite cui un attaccante, installando un Virtual Machine Monitor sul sistema attaccato, può creare dei servizi a livello di VMM per controllare lo stato delle macchine virtuali. IntroVirt [56] è un sistema che, tramite l'uso di predicati per ricercare vulnerabilità, di tecniche d'introspezione e di *virtual-machine replay* riesce a rilevare intrusioni che sono avvenute in un momento passato della storia della macchina virtuale osservata. Collapsar [55] è un sistema che viene eseguito su una rete di macchine virtuali, e ha il compito di coordinare un insieme di *honeypot* installati su reti logiche differenti per il rilevamento di attacchi di rete. Infine, HyperSpector [61] è un sistema di IDS distribuito su una rete di macchine virtuali.

Conclusioni

> "About all I know is, I sort of *miss* everybody I told about. Even old Stradlater and Ackley, for instance. I think I even miss that goddam Maurice. It's funny. Don't ever tell anybody anything. If you do, you start missing everybody."
>
> *The Catcher in the Rye*, 1951
> J. D. Salinger

Il sistema Psyco-Virt unisce le tecnologie di virtualizzazione e le tecniche per il controllo dell'esecuzione dei sistemi sulle macchine virtuali con le normali metodologie di rilevamento delle intrusioni, per fornire un'architettura in grado di rilevare e prevenire le intrusioni su ambienti virtuali, monitorando il sistema a più livelli.

Una delle principali innovazioni di Psyco-Virt è l'utilizzo di tecniche di introspezione su un'architettura distribuita di IDS, e la presenza di una rete di controllo utilizzata sia per coordinare le varie componenti sulle macchine virtuali che per inviare alert relativi ad intrusioni e attacchi. Un'altra novità è rappresentata dalla possibilità di monitorare le componenti stesse del sistema in esecuzione sulle macchine virtuali, quali il collector e gli agent, per prevenire modifiche effettuate da un attaccante all'IDS stesso.

In questo modo, sebbene i collector e gli agent siano in esecuzione in un ambiente "ostile", e quindi se un attaccante è riuscito a compromettere l'host in esecuzione nella macchina virtuale e ha modificato i collector e gli agent per fornire, tramite essi, informazioni false sullo stato del sistema monitorato, l'engine è comunque in grado di rilevare tali modifiche effettuate dall'attaccante grazie alle tecniche di introspezione applicate sui collector e sugli agent.

Inoltre, questa architettura permette di mitigare in parte il problema dei falsi positivi:

infatti, dato che in caso di intrusioni e attacchi si possono eseguire azioni automatiche sulla macchina virtuale monitorata, come sospenderne l'esecuzione o salvarne il contenuto della memoria su un file, è possibile eseguire successivamente a queste azioni controlli più accurati sul sistema monitorato per verificare se siamo in presenza di intrusioni reali o falsi positivi. Lo stato dell'host monitorato viene così "congelato" e ripristinato successivamente se, dopo aver effettuato ulteriori controlli, non vengono rilevate compromissioni.

In questa maniera, i falsi positivi non sono più avvertiti dall'amministratore di sistema con un senso di sfiducia verso l'IDS. Infatti, basta associare agli eventi che possono generare i falsi positivi azioni automatiche di sospensione delle macchine virtuali. Facendo così, l'amministratore non è costretto ad abbassare il livello di monitoraggio per cercare di diminuire il numero dei falsi positivi.

Il prototipo realizzato in Python per i test ha dimostrato la fattibilità dell'architettura proposta. Il prototipo ha anche messo in evidenza i vantaggi che si ottengono utilizzando le tecniche di introspezione in combinazione con gli usuali strumenti di rilevamento delle intrusioni, per effettuare controlli a più livelli. In questa maniera, se i dati ottenuti da diverse fonti, ad esempio i dati ottenuti dagli agenti e inviati sulla rete di controllo e quelli ottenuti dai moduli tramite introspezione, sono differenti, allora l'host monitorato è stato compromesso da un attaccante per fornire dati falsi. In questo caso, si possono eseguire azioni in risposta all'intrusione, ad esempio sospendere l'esecuzione della macchina virtuale compromessa.

Appendice A

Codice

La versione utilizzata dell'interprete di Python è la 2.4.3.

main.py

```
import gui
import Tkinter

def main():
    #create the root Tkinter
    root = Tkinter.Tk()
    gui.Gui(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

gui.py

```
from Tkinter import *
import os
from loadConf import splitChar
import loadConf
import engine
import actions
import saveConf
import logging
import Queue
import FileDialog
import tkSimpleDialog

class processMsg:

    def __init__(self, gui):
        self.gui=gui
        self.queue=gui.queue
        self.root=gui.root
        self.running = 1
        self.periodicCall()
```

```
        def periodicCall(self):
            self.gui.processIncoming()
            if not self.running:
                import sys
                sys.exit(1)
            self.root.after(100, self.periodicCall)

        def write(self, msg):
            self.queue.put(msg)
            pass

        def flush(self):
            pass

    class Gui:
        def __init__(self, root):
            self.loadconf = loadConf.LoadConf()
            tkSimpleDialog.loadconf = self.loadconf
            self.myName = self.__class__.__name__
            self.root = root
            self.eng = engine.Engine(self.loadconf)
            self.loaded = 0
            self.running = 0
            self.oldAppLogName = ''
            self.queue = Queue.Queue()
            #------ constants for controlling layout of buttons ------
            self.button_width = 10
            self.button_height = 1
            self.button_padx = "2m"
            self.button_pady = "1m"
            self.buttons_frame_padx =  "3m"
            self.buttons_frame_pady =  "0.2m"
            self.buttons_frame_ipadx = "3m"
            self.buttons_frame_ipady = "0.2m"
            self.listBoxWidth = 25
            self.listBoxHeight = 5
            # ---------------- end constants -----------------
            self.configurationFile = ''

            #for saving configuration
            self.XMLfile = ''
            self.XMLlogging = {}
            self.XMLmodules = {}
            self.XMLlevels = {}
            self.XMLactions = {}
            self.XMLhosts = {}
            self.XMLparameters = {}

            self.loggingLevels = {}
            self.defaultLoggingNames = {}
            for a, b in logging._levelNames.items():
                self.defaultLoggingNames[a] = b
            self.consoleLevel = IntVar()
            self.idsLogFileLevel = IntVar()
            self.appLogFileLevel  = IntVar()

            self.root.geometry('950x650')
            self.root.minsize('950','650')
            self.root.title('Psyco-Virt')
            #self.root.resizable(NO, NO)
            self.createMainFrame()
            self.createLeftFrame()
            self.createRightFrame()
            self.createMenu()
            self.loadDefaults()
            self.createConsoleLogger()
            self.createAppLogger()
            self.log(loadConf.START, self.myName + ' started')
```

```
    def createModuleDialog(self):
        d = tkSimpleDialog.moduleDialog(self.root, title='Add a module')
        if d is not None:
            if d.module is not '' and d.className is not '':
                line = '%s, %s' % (d.module, d.className)
                self.modulesListBox.insert(END, line)

    def createVMDialog(self):
        d = tkSimpleDialog.vmDialog(self.root, title='Add a VM')
        if d is not None:
            if d.vmName is not '' and d.vmIp is not '' and d.vmPort is not '':
                line = '%s, %s:%s' % (d.vmName, d.vmIp, d.vmPort)
                self.vmListBox.insert(END, line)
        else:
            print 'vm is null'

    def createServerDialog(self):
        d = tkSimpleDialog.serverDialog(self.root, title='Add the server')
        if d is not None:
            if d.serverIp is not '' and d.serverPort is not '':
                line = '%s:%s' % (d.serverIp, d.serverPort)
                self.serverText.config(state=NORMAL)
                self.serverText.delete(0, END)
                self.serverText.insert(END, line)
                self.serverText.config(state=DISABLED)

    def createActionDialog(self):
        d = tkSimpleDialog.actionDialog(self.root, loggingLevels=self.loggingLevels,
                                        title='Add an action to a level')
        if d is not None:
            if d.module is '':
                if d.level is not '' and d.action is not '':
                    line = '%s, %s, %s' % (d.level, d.action, 'default')
                    self.actionsListBox.insert(END, line)
            else:
                if d.level is not '' and d.action is not '':
                    line = '%s, %s, %s, %s' % (d.level, d.action, 'custom', d.module)
                    self.actionsListBox.insert(END, line)

    def createLevelDialog(self):
        d = tkSimpleDialog.levelDialog(self.root, title='Add a new level')
        if d is not None:
            if d.levelGravity is not '' and d.levelName is not '':
                self.loggingLevels[int(d.levelGravity)] = d.levelName
                self.writeLevels()

    def createIdsLogFileDialog(self):
        fdlg = FileDialog.SaveFileDialog(self.root, title="Choose A File")
        filename = fdlg.go(pattern="*.log")
        if filename is not None:
            line = '%s' % filename.strip()
            self.idsLogFileText.config(state=NORMAL)
            self.idsLogFileText.delete(0, END)
            self.idsLogFileText.insert(END, line)
            self.idsLogFileText.xview(END)
            self.idsLogFileText.config(state=DISABLED)

    def createAppLogFileDialog(self):
        fdlg = FileDialog.SaveFileDialog(self.root, title="Choose A File")
        filename = fdlg.go(pattern="*.log")
        if filename is not None:
            line = '%s' % filename.strip()
            self.appLogFileText.config(state=NORMAL)
            self.appLogFileText.delete(0, END)
            self.appLogFileText.insert(END, line)
            self.appLogFileText.xview(END)
            self.appLogFileText.config(state=DISABLED)

    def deleteSelectedModule(self):
```

```
        items = self.moduleDialog.curselection()

    def deleteLevel(self):
        line = self.levelsListBox.get(ANCHOR)
        gravity, level = line.split(':')
        gravity = int(gravity)
        tmp = {}
        for a, b in self.loggingLevels.items():
            if a != gravity:
                tmp[a] = b
        self.loggingLevels = {}
        for a, b in tmp.items():
            self.loggingLevels[a] = b
        self.writeLevels()

    def writeLevels(self):
        self.levelsListBox.delete(0, END)
        self.idsLogFileMenu.menu.delete(0, END)
        self.appLogFileMenu.menu.delete(0, END)
        self.consoleLogMenu.menu.delete(0, END)
        #write log levels to menus
        values = self.loggingLevels.keys()
        values.sort()
        self.consoleLevel.set(self.loadconf.consoleLevel)
        self.idsLogFileLevel.set(self.loadconf.idsLogFileLevel)
        self.appLogFileLevel.set(self.loadconf.appLogFileLevel)
        for value in values:
            if value < self.loadconf.applicationStartingLevel:
                self.idsLogFileMenu.menu.add_radiobutton(
                    label='%s:%s'%(value, self.loggingLevels[value]),
                    variable=self.idsLogFileLevel,
                    value=value)
                self.levelsListBox.insert(END, '%s:%s'%(value, self.loggingLevels[value]))
            else:
                self.appLogFileMenu.menu.add_radiobutton(
                    label='%s:%s'%(value, self.loggingLevels[value]),
                    variable=self.appLogFileLevel,
                    value=value)
                self.consoleLogMenu.menu.add_radiobutton(
                    label='%s:%s'%(value, self.loggingLevels[value]),
                    variable=self.consoleLevel,
                    value=value)

    def loadDefaultLevels(self):
        values = logging._levelNames.items()
        for gravity, name in values:
            if type(gravity) == type(0):
                self.loggingLevels[gravity] = name
        for gravity, name in self.loadconf.levels.items():
            if type(gravity) == type(0):
                self.loggingLevels[gravity] = name

    def loadLevels(self):
        for gravity, name in self.loadconf.levels.items():
            if type(gravity) == type(0):
                self.loggingLevels[gravity] = name

    def addLevelNames(self):
        for levelName, gravity in self.loggingLevels.items():
            logging.addLevelName(gravity, levelName)

    def loadDefaults(self):
        self.clearAll()
        self.loadconf.resetToDefault()
        self.resetLoggingLevels()
        self.loadDefaultLevels()
        self.addLevelNames()
        self.writeLevels()
        self.serverText.config(state=NORMAL)
```

```
229             self.serverText.insert(END, '%s:%s' % (self.loadconf.serverDefaultIp, self.loadconf.serverDefaultPort))
                self.serverText.config(state=DISABLED)
                self.idsLogFileText.config(state=NORMAL)
                self.idsLogFileText.insert(END, self.loadconf.idsLogFile)
                self.idsLogFileText.config(state=DISABLED)
234             self.appLogFileText.config(state=NORMAL)
                self.appLogFileText.insert(END, self.loadconf.appLogFile)
                self.appLogFileText.config(state=DISABLED)

            def log(self, name, msg):
239             level = self.loadconf.levels[name]
                self.appLogger.log(level, msg)
                self.consoleLogger.log(level, msg)

            def processIncoming(self):
244             """
                Handle all the messages currently in the queue (if any).
                """
                while self.queue.qsize():
                    try:
249                     line = self.queue.get(0)
                        # Check contents of message and do what it says
                        time, name, level, msg = line.split(splitChar) #log msg is: time name level msg
                        self.consoleText.insert(END,'%s%s%s%s' % (time, name, level, msg))
                    except Queue.Empty:
254                     pass

            def resetLoggingLevels(self):
                logging._levelNames = {}
                for a, b in self.defaultLoggingNames.items():
259                 logging._levelNames[a] = b

            def createConsoleLogger(self):
                self.pm = processMsg(self)
                consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName)
264             self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName)
                self.consoleLogger.setLevel(1)
                self.formatter = logging.Formatter(
                    '%(asctime)s '+splitChar+' %(name)s '+splitChar+' %(levelname)s '+splitChar+' %(message)s')
                ch = logging.StreamHandler(self.pm)
269             ch.setLevel(self.loadconf.appLogFileLevel)
                ch.setFormatter(self.formatter)
                self.consoleLogger.addHandler(ch)

            def createAppLogger(self):
274             #if name isn't changed, DONT create the logger (or you get the same message twice)
                open(self.loadconf.appLogFile, 'w')
                if self.oldAppLogName != self.loadconf.appLogRootName:
                    self.appLogger = logging.getLogger(self.loadconf.appLogRootName)
                    self.appLogger.setLevel(1)
279                 afh = logging.FileHandler(self.loadconf.appLogFile)
                    afh.setLevel(self.loadconf.appLogFileLevel)
                    afh.setFormatter(self.formatter)
                    self.appLogger.addHandler(afh)
                    self.oldAppLogName = self.loadconf.appLogRootName
284

            def clearAll(self):
                self.loggingLevels = {}
                self.modulesListBox.delete(0, END)
289             self.vmListBox.delete(0, END)
                self.actionsListBox.delete(0, END)
                self.levelsListBox.delete(0, END)
                self.idsLogFileText.config(state=NORMAL)
                self.idsLogFileText.delete(0, END)
294             self.idsLogFileText.config(state=DISABLED)
                self.serverText.config(state=NORMAL)
                self.serverText.delete(0, END)
                self.serverText.config(state=DISABLED)
```

```
            self.appLogFileText.config(state=NORMAL)
            self.appLogFileText.delete(0, END)
            self.appLogFileText.config(state=DISABLED)
            self.idsLogFileMenu.menu.delete(0, END)
            self.appLogFileMenu.menu.delete(0, END)
            self.consoleLogMenu.menu.delete(0, END)
            self.configurationFileText.config(state=NORMAL)
            self.configurationFileText.delete(0, END)
            self.configurationFileText.config(state=DISABLED)

        def loadConf(self, filename=None):
            if filename == None:
                fdlg = FileDialog.LoadFileDialog(self.root, title="Choose A File")
                filename = fdlg.go(pattern="*.xml")
                if filename == None:
                    return
            self.loadconf.resetToDefault()
            self.loadconf.setConfFile(filename)
            self.loadconf.load()
            self.configurationFile = filename
            self.clearAll()
            self.loadLevels()
            self.addLevelNames()
            self.loaded = 1
            self.log(loadConf.LOAD, "Configuration loaded: %s" % filename)

            #write modules to gui
            for item in self.loadconf.modules:
                line = '%s, %s' % (item[0], item[1])
                self.modulesListBox.insert(END, line)
            #write vm to gui
            for item in self.loadconf.vm.items():
                line = '%s, %s:%s' %(item[0], item[1][0], item[1][1])
                self.vmListBox.insert(END, line)
            self.serverText.config(state=NORMAL)
            self.serverText.insert(END, '%s:%s'%(self.loadconf.serverIp, self.loadconf.serverPort))
            self.serverText.config(state=DISABLED)
            #write actions to gui
            for item in self.loadconf.levelsActions.items():
                if item[1][1] == 'default':
                    line = '%s, %s, %s' % (item[0], item[1][0], item[1][1])
                else:
                    line = '%s, %s, %s, %s' % (item[0], item[1][0], item[1][1], item[1][2])
                self.actionsListBox.insert(END, line)
            #write log files to gui
            self.idsLogFileText.config(state=NORMAL)
            self.idsLogFileText.insert(END, self.loadconf.idsLogFile)
            self.idsLogFileText.config(state=DISABLED)
            self.appLogFileText.config(state=NORMAL)
            self.appLogFileText.insert(END, self.loadconf.appLogFile)
            self.appLogFileText.config(state=DISABLED)
            self.configurationFileText.config(state=NORMAL)
            self.configurationFileText.insert(END, self.configurationFile)
            self.configurationFileText.xview(END)
            self.configurationFileText.config(state=DISABLED)
            self.writeLevels()
            return filename

        def startEngine(self):
            if self.loaded and not self.running:
                self.eng.start()
                self.running = 1

        def stopEngine(self):
            if self.running:
                self.eng.stop()
                self.running = 0

        def quit(self):
```

```
            self.log(loadConf.STOP, self.myName + ' stopped')
            self.stopEngine()
            self.mainFrame.quit()

        def applyChanges(self):
            filename=self.saveConf()
            self.self.loadconf(filename)
            self.stopEngine()
            self.startEngine()

        def XMLsaveLogging(self):
            #ids logfile
            self.XMLlogging['logFile'] = [{'type': 'ids', 'name':(self.idsLogFileText.get()).strip(),
                                           'level': self.idsLogFileLevel.get()}]
            #appl logfile
            newElem = self.XMLlogging['logFile']
            newElem.append({'type': 'application', 'name':(self.appLogFileText.get()).strip(),
                            'level': self.appLogFileLevel.get()})
            #console log
            self.XMLlogging['logFile'] = newElem
            newElem = self.XMLlogging['logFile']
            newElem.append({'type': 'console', 'level': self.consoleLevel.get()})

        def XMLsaveModules(self):
            modules = self.modulesListBox.get(0, END)
            if modules is not None:
                for item in modules:
                    name, className = item.split(',')
                    name = name.strip()
                    className = className.strip()
                    if self.XMLmodules.has_key('module'):
                        newElem = self.XMLmodules['module']
                        newElem.append({'name': name, 'class' : className})
                        self.XMLmodules['module'] = newElem
                    else:
                        self.XMLmodules['module'] = [{'name': name, 'class' : className}]

        def XMLsaveHosts(self):
            vm = self.vmListBox.get(0, END)
            if vm is not None:
                for item in vm:
                    vmName, vmIpPort = item.split(',')
                    vmName = vmName.strip()
                    vmIp, vmPort = vmIpPort.split(':')
                    vmIp = vmIp.strip()
                    vmPort = vmPort.strip()
                    if self.XMLhosts.has_key('vm'):
                        newElem = self.XMLhosts['vm']
                        newElem.append({'name': vmName, 'ip' : vmIp, 'port' : vmPort})
                        self.XMLhosts['vm'] = newElem
                    else:
                        self.XMLhosts['vm'] = [{'name': vmName, 'ip' : vmIp, 'port' : vmPort}]
            #solo un server
            if(self.serverText.get() is not ''):
                serverIp, serverPort = ((self.serverText.get()).strip()).split(':')
                self.XMLhosts['server'] = [{'ip': serverIp, 'port' : serverPort}]

        def XMLsaveActions(self):
            actions = self.actionsListBox.get(0, END)
            if actions is not None:
                for item in actions:
                    values = item.split(',')
                    if len(values) == 4:
                        module = values[3].strip()
                    level = values[0].strip()
                    action = values[1].strip()
                    type = values[2].strip()
                    if type == 'default':
                        if self.XMLactions.has_key('action'):
```

```
                        newElem = self.XMLactions['action']
                        newElem.append({'level': level, 'execute' : action, 'type' : type})
                        self.XMLactions['action'] = newElem
439                 else:
                        self.XMLactions['action'] = [{'level': level, 'execute' : action,
                                                      'type' : type}]
                elif type == 'custom':
                    if self.XMLactions.has_key('action'):
444                     newElem = self.XMLactions['action']
                        newElem.append({'level': level, 'execute' : action,
                                        'type' : type, 'module' : module})
                        self.XMLactions['action'] = newElem
                    else:
449                     self.XMLactions['action'] = [{'level': level, 'execute' : action,
                                                      'type' : type, 'module' : module}]

    def XMLsaveLevels(self):
        levels = self.levelsListBox.get(0, END)
454     if levels is not None:
            for item in levels:
                gravity, name = item.split(':')
                gravity = gravity.strip()
                name = name.strip()
459             if self.XMLlevels.has_key('level'):
                    newElem = self.XMLlevels['level']
                    newElem.append({'gravity': gravity, 'name' : name})
                    self.XMLlevels['level'] = newElem
                else:
464                 self.XMLlevels['level'] = [{'gravity': gravity, 'name' : name}]



    def saveConf(self):
469     #obbligatorio niente
        fdlg = FileDialog.SaveFileDialog(self.root, title="Choose A File")
        filename = fdlg.go(pattern="*.xml")
        if filename == None:
            return
474     else:
            self.XMLfile = filename
            self.XMLsaveLogging()
            self.XMLsaveModules()
            self.XMLsaveHosts()
479         self.XMLsaveActions()
            self.XMLsaveLevels()
            s = saveConf.SaveConf(self.loadconf, self.XMLfile, self.XMLlogging, self.XMLmodules,
                                  self.XMLlevels, self.XMLactions, self.XMLhosts, self.XMLparameters)
            s.writeAll()
484         self.XMLlogging = {}
            self.XMLmodules = {}
            self.XMLlevels = {}
            self.XMLactions = {}
            self.XMLhosts = {}
489         self.XMLparameters = {}
            return filename

    def createMenu(self):
        self.menu = Menu()
494     self.root.config(menu=self.menu)
        self.filemenu = Menu(self.menu)
        self.menu.add_cascade(label="File", menu=self.filemenu)
        self.filemenu.add_command(label="New", command=self.loadDefaults)
        self.filemenu.add_command(label="Open...", command=self.loadConf)
499     self.filemenu.add_command(label="Save...", command=self.saveConf)
        self.filemenu.add_separator()
        self.filemenu.add_command(label="Exit", command=self.quit)
        self.helpmenu = Menu(self.menu)
        self.menu.add_cascade(label="Help", menu=self.helpmenu)
504
```

```
    def createMainFrame(self):
        self.mainFrame = Frame(self.root)
        self.mainFrame.pack(expand=YES, fill=BOTH)

509
    def createLeftFrame(self):
        self.leftFrame = Frame(self.mainFrame)
        self.leftFrame.pack(side=LEFT, expand=NO, fill=BOTH, padx=10, pady=5, ipadx=5, ipady=5)
        #MODULES#
514     #grid
        #[row0][col0]:label
        #[row1][col0]:frame(listBox, scrollbar) [col1]:frame(grid:[row0]buttonAdd, [row1]buttonDelete)

        #[row0][col0]label
519     self.leftLastRow=0
        Label(self.leftFrame, text='MODULES\nname, class',
              justify=LEFT).grid(row=self.leftLastRow, column=0, sticky=W+N)
        self.leftLastRow+=1
        #[row1][col0]:frame
524     self.modulesFrame = Frame(self.leftFrame)
        self.modulesFrame.grid(row=self.leftLastRow, column=0, sticky=N)
        self.modulesXScrollbar = Scrollbar(self.modulesFrame, orient=HORIZONTAL)
        self.modulesYScrollbar = Scrollbar(self.modulesFrame, orient=VERTICAL)
        self.modulesListBox = Listbox(self.modulesFrame,
529                                   relief=SUNKEN,
                                      selectborderwidth=0, selectmode=BROWSE,
                                      bg='white',
                                      xscrollcommand=self.modulesXScrollbar.set,
                                      yscrollcommand=self.modulesYScrollbar.set,
534                                   width=self.listBoxWidth, height=self.listBoxHeight)
        self.modulesXScrollbar.config(command=self.modulesListBox.xview)
        self.modulesYScrollbar.config(command=self.modulesListBox.yview)
        self.modulesXScrollbar.pack(side=BOTTOM, fill=X)
        self.modulesYScrollbar.pack(side=RIGHT, fill=Y)
539     self.modulesListBox.pack(side=LEFT, fill=BOTH, expand=YES)
        #[row1][col1]:frame
        self.modulesButtonsFrame = Frame(self.leftFrame)
        self.modulesButtonsFrame.grid(row=self.leftLastRow, column=1, sticky=W+N)
        self.leftLastRow+=1
544     self.modulesButtonAdd = Button(self.modulesButtonsFrame, text="add",
                                       height=self.button_height,
                                       width=self.button_width, padx=self.button_padx,
                                       pady=self.button_pady, command=self.createModuleDialog)
        self.modulesButtonRemove = Button(self.modulesButtonsFrame, text="remove",
549                                       height=self.button_height,
                                          width=self.button_width, padx=self.button_padx,
                                          pady=self.button_pady,
                                          command=lambda lb=self.modulesListBox: lb.delete(ANCHOR))
        self.modulesButtonAdd.grid(row=0, column=0, sticky=W+N)
554     self.modulesButtonRemove.grid(row=1, column=0, sticky=W+N)

        #VIRTUAL MACHINES
        #[row0][col0]label
        Label(self.leftFrame, text='VM\nname, ip:port', justify=LEFT).grid(row=self.leftLastRow,
559                                                                        column=0, sticky=W+N)
        self.leftLastRow+=1
        #[row1][col0]:frame
        self.vmFrame = Frame(self.leftFrame)
        self.vmFrame.grid(row=self.leftLastRow, column=0, sticky=W+N)
564     self.vmXScrollbar = Scrollbar(self.vmFrame, orient=HORIZONTAL)
        self.vmYScrollbar = Scrollbar(self.vmFrame, orient=VERTICAL)
        self.vmListBox = Listbox(self.vmFrame, bg='white', xscrollcommand=self.vmXScrollbar.set,
                                 yscrollcommand=self.vmYScrollbar.set,
                                 width=self.listBoxWidth, height=self.listBoxHeight)
569     self.vmXScrollbar.config(command=self.vmListBox.xview)
        self.vmYScrollbar.config(command=self.vmListBox.yview)
        self.vmXScrollbar.pack(side=BOTTOM, fill=X)
        self.vmYScrollbar.pack(side=RIGHT, fill=Y)
        self.vmListBox.pack(side=LEFT, fill=BOTH, expand=YES)
```

```
574             #[row1][col1]:frame
                self.vmButtonsFrame = Frame(self.leftFrame)
                self.vmButtonsFrame.grid(row=self.leftLastRow, column=1, sticky=W+N)
                self.leftLastRow+=1
                self.vmButtonAdd = Button(self.vmButtonsFrame, text="add",
579                                 height=self.button_height,
                                    width=self.button_width, padx=self.button_padx,
                                    pady=self.button_pady, command=self.createVMDialog)
                self.vmButtonRemove = Button(self.vmButtonsFrame, text="remove",
                                        height=self.button_height,
584                                     width=self.button_width, padx=self.button_padx,
                                        pady=self.button_pady,
                                        command=lambda lb=self.vmListBox: lb.delete(ANCHOR))
                self.vmButtonAdd.grid(row=0, column=0, sticky=W+N)
                self.vmButtonRemove.grid(row=1, column=0, sticky=W+N)
589
                #SERVER
                Label(self.leftFrame, text='SERVER\nip:port', justify=LEFT).grid(row=self.leftLastRow,
                                                                                 column=0, sticky=W+N)
                self.leftLastRow+=1
594             self.serverText = Entry(self.leftFrame, bg='white', width=self.listBoxWidth)
                self.serverText.config(state=DISABLED)
                self.serverText.grid(row=self.leftLastRow, column=0, sticky=W)
                #change button
                self.serverButton = Button(self.leftFrame, text='change', width=self.button_width,
599                                 height=self.button_height,
                                    padx=self.button_padx, pady=self.button_pady,
                                    command=self.createServerDialog)
                self.serverButton.grid(row=self.leftLastRow, column=1, sticky=W)
                self.leftLastRow+=1
604
                #ACTIONS
                #grid
                #[row0][col0]label
                Label(self.leftFrame, text='ACTIONS\nlevel, action, type, [class]',
609                   justify=LEFT).grid(row=self.leftLastRow, column=0, sticky=W+S)
                self.leftLastRow+=1
                #[row1][col0]:frame
                self.actionsFrame = Frame(self.leftFrame)
                self.actionsFrame.grid(row=self.leftLastRow, column=0, sticky=W+N)
614             self.actionsXScrollbar = Scrollbar(self.actionsFrame, orient=HORIZONTAL)
                self.actionsYScrollbar = Scrollbar(self.actionsFrame, orient=VERTICAL)
                self.actionsListBox = Listbox(self.actionsFrame, xscrollcommand=self.actionsXScrollbar.set,
                                              yscrollcommand=self.actionsYScrollbar.set,
                                              bg='white', width=self.listBoxWidth, height=self.listBoxHeight)
619             self.actionsXScrollbar.config(command=self.actionsListBox.xview)
                self.actionsYScrollbar.config(command=self.actionsListBox.yview)
                self.actionsXScrollbar.pack(side=BOTTOM, fill=X)
                self.actionsYScrollbar.pack(side=RIGHT, fill=Y)
                self.actionsListBox.pack(side=LEFT, fill=BOTH, expand=YES)
624             #[row1][col1]:frame
                self.actionsButtonsFrame = Frame(self.leftFrame)
                self.actionsButtonsFrame.grid(row=self.leftLastRow, column=1, sticky=W+N)
                self.leftLastRow+=1
                self.actionsButtonAdd = Button(self.actionsButtonsFrame, text="add",
629                                     width=self.button_width, padx=self.button_padx,
                                        pady=self.button_pady, command=self.createActionDialog)
                self.actionsButtonRemove = Button(self.actionsButtonsFrame, text="remove",
                                            width=self.button_width, padx=self.button_padx,
                                            pady=self.button_pady,
634                                         command=lambda lb=self.actionsListBox: lb.delete(ANCHOR))
                self.actionsButtonAdd.grid(row=0, column=0, sticky=W+N)
                self.actionsButtonRemove.grid(row=1, column=0, sticky=W+N)

                #LEVELS
639             Label(self.leftFrame, text='LEVELS\nlevelGravity:levelName',
                      justify=LEFT).grid(row=self.leftLastRow, column=0, sticky=W+S)
                self.leftLastRow+=1
                #[row1][col0]:frame
```

```
        self.levelsFrame = Frame(self.leftFrame)
644     self.levelsFrame.grid(row=self.leftLastRow, column=0, sticky=W+N)
        self.levelsXScrollbar = Scrollbar(self.levelsFrame, orient=HORIZONTAL)
        self.levelsYScrollbar = Scrollbar(self.levelsFrame, orient=VERTICAL)
        self.levelsListBox = Listbox(self.levelsFrame, xscrollcommand=self.levelsXScrollbar.set,
                                     yscrollcommand=self.levelsYScrollbar.set,
649                                  bg='white', width=self.listBoxWidth, height=self.listBoxHeight)
        self.levelsXScrollbar.config(command=self.levelsListBox.xview)
        self.levelsYScrollbar.config(command=self.levelsListBox.yview)
        self.levelsXScrollbar.pack(side=BOTTOM, fill=X)
        self.levelsYScrollbar.pack(side=RIGHT, fill=Y)
654     self.levelsListBox.pack(side=LEFT, fill=BOTH, expand=YES)
        #[row1][col1]:frame
        self.levelsButtonsFrame = Frame(self.leftFrame)
        self.levelsButtonsFrame.grid(row=self.leftLastRow, column=1, sticky=W+N)
        self.leftLastRow+=1
659     self.levelsButtonAdd = Button(self.levelsButtonsFrame, text="add",
                                      width=self.button_width, padx=self.button_padx,
                                      pady=self.button_pady, command=self.createLevelDialog)
        self.levelsButtonRemove = Button(self.levelsButtonsFrame, text="remove",
                                         width=self.button_width, padx=self.button_padx,
664                                      pady=self.button_pady, command=self.deleteLevel)
        self.levelsButtonAdd.grid(row=0, column=0, sticky=W+N)
        self.levelsButtonRemove.grid(row=1, column=0, sticky=W+N)

    def createRightFrame(self):
669
        self.rightFrame = Frame(self.mainFrame)
        self.rightFrame.pack(side=RIGHT, expand=YES, fill=BOTH, padx=10, pady=5, ipadx=5, ipady=5)

        self.rightLastRow=0
674     #right top frame
        #grid
        #[row0][col0]:frame:console
        #[row1][col0]:frame:buttons
        #[row2][col0]:grid: (row[0]col[0]: label, row[0]col[1]: text, row[1]col[0]: label, row[1]col[1]: text)
679     #[row3][col0]:frame (buttons)

        #[row0][col0]:console
        self.rightFrame.grid_rowconfigure(0, weight=1)
        self.rightFrame.grid_columnconfigure(0, weight=1)
684     self.consoleFrame = Frame(self.rightFrame)
        self.consoleFrame.grid(row=self.rightLastRow, column=0, columnspan=2,sticky=W+N+E+S)
        self.rightLastRow+=1
        self.consoleYScrollbar = Scrollbar(self.consoleFrame, orient=VERTICAL)
        self.consoleText = Text(self.consoleFrame, bg='white',
689                             font='Times 13',
                            yscrollcommand=self.consoleYScrollbar.set)
        self.consoleYScrollbar.config(command=self.consoleText.yview)
        self.consoleYScrollbar.pack(side=RIGHT, fill=Y, expand=NO)
        self.consoleText.pack(side=LEFT, fill=BOTH, expand=YES, anchor=W)
694
        #[row1][col0]:frame:button+menu
        self.consoleButtonsFrame = Frame(self.rightFrame)
        self.consoleButtonsFrame.grid(row=self.rightLastRow, column=0, sticky=W+N+E+S)
        self.rightLastRow+=1
699
        self.consoleButtonClear = Button(self.consoleButtonsFrame, text="clear",
                                         width=self.button_width, height=self.button_height,
                                         padx=self.button_padx,
                                         pady=self.button_pady,
704                                      command=lambda tc=self.consoleText: tc.delete(1.0, END))
        self.consoleButtonClear.pack(side=LEFT)

        self.configurationFileText = Entry(self.consoleButtonsFrame,
                                           bg='white', font='*-Courier-Bold-R-Normal-*-120-*')
709     self.configurationFileText.config(state=DISABLED)
        self.configurationFileText.pack(side=RIGHT)
        Label(self.consoleButtonsFrame,
```

```
            text='configuration file: ',
            justify=LEFT, relief=FLAT, anchor=NW, borderwidth=0,
            font='*-Courier-Bold-R-Normal-*-120-*').pack(side=RIGHT)

        #menu
        self.consoleLogMenu = Menubutton(self.consoleButtonsFrame, height=self.button_height,
                                         width=self.button_width,
                                         padx=self.button_padx, pady=self.button_pady,
                                         text='Base level',
                                         relief=RAISED )
        self.consoleLogMenu.pack(side=LEFT)
        self.consoleLogMenu.menu = Menu(self.consoleLogMenu, tearoff=0 )
        self.consoleLogMenu["menu"] = self.consoleLogMenu.menu

        #second row
        #[row2][col0]:grid:
        #log frame
        #first row
        self.logFrame = Frame(self.rightFrame)
        self.logFrame.grid(row=self.rightLastRow, column=0, sticky=W+N+E+S)
        self.rightLastRow+=1
        Label(self.logFrame,
            text='log file: ',
            justify=LEFT, relief=FLAT, anchor=NW, borderwidth=0,
            font='*-Courier-Bold-R-Normal-*-120-*').grid(row=0, column=0, sticky=W)
        self.idsLogFileText = Entry(self.logFrame, bg='white')
        self.idsLogFileText.config(state=DISABLED)
        self.idsLogFileText.grid(row=0, column=1)
        #change button
        self.idsLogFileButton = Button(self.logFrame, text='change', height=self.button_height,
                                       width=self.button_width ,
                                       padx=self.button_padx, pady=self.button_pady,
                                       command=self.createIdsLogFileDialog)
        self.idsLogFileButton.grid(row=0, column=2, sticky=W)
        #menu
        self.idsLogFileMenu = Menubutton(self.logFrame, text="Base level", height=self.button_height,
                                      width=self.button_width,
                                      padx=self.button_padx, pady=self.button_pady,
                                      relief=RAISED)
        self.idsLogFileMenu.grid(row=0, column=3, sticky=W)
        self.idsLogFileMenu.menu = Menu(self.idsLogFileMenu, tearoff=0 )
        self.idsLogFileMenu["menu"] = self.idsLogFileMenu.menu
        #second row
        Label(self.logFrame,
            text='application log file: ',
            justify=LEFT, relief=FLAT, anchor=NW, borderwidth=0,
            font='*-Courier-Bold-R-Normal-*-120-*').grid(row=1, column=0, sticky=W)
        self.appLogFileText = Entry(self.logFrame, bg='white')
        self.appLogFileText.config(state=DISABLED)
        self.appLogFileText.grid(row=1, column=1)
        #changeButton
        self.appLogFileButton = Button(self.logFrame, text='change', width=self.button_width,
                                       height=self.button_height,
                                       padx=self.button_padx, pady=self.button_pady,
                                       command=self.createAppLogFileDialog)
        self.appLogFileButton.grid(row=1, column=2, sticky=W)
        #menu
        self.appLogFileMenu = Menubutton(self.logFrame, text="Base level", height=self.button_height,
                                         width=self.button_width,
                                         padx=self.button_padx, pady=self.button_pady,
                                         relief=RAISED)
        self.appLogFileMenu.grid(row=1, column=3, sticky=W)
        self.appLogFileMenu.menu = Menu(self.appLogFileMenu, tearoff=0 )
        self.appLogFileMenu["menu"] = self.appLogFileMenu.menu


        #[row3][col0]:frame (buttons)
        #buttons frame
        self.buttonsFrame = Frame(self.rightFrame)
```

```
        self.buttonsFrame.grid(row=self.rightLastRow, column=0, sticky=W+N+E+S)
        self.rightLastRow+=1
        self.buttonStart = Button(self.buttonsFrame, text="start", fg="red",
784                               width=self.button_width, padx=self.button_padx,
                                  pady=self.button_pady, command=self.startEngine)
        self.buttonStart.pack(side=LEFT)

        self.buttonApply = Button(self.buttonsFrame, text="apply",
789                                   width=self.button_width, padx=self.button_padx,
                                      pady=self.button_pady, command=self.applyChanges)
        self.buttonApply.pack(side=LEFT)


794
        self.buttonQuit = Button(self.buttonsFrame, text="quit",
                                  width=self.button_width, padx=self.button_padx,
                                  pady=self.button_pady, command=self.quit)
        self.buttonQuit.pack(side=LEFT)
799     self.buttonConf = Button(self.buttonsFrame, text="suspend",
                                  width=self.button_width, padx=self.button_padx,
                                  pady=self.button_pady, command=self.eng.config)
        self.buttonConf.pack(side=LEFT)
        self.buttonPs = Button(self.buttonsFrame, text="resume",
804                            width=self.button_width, padx=self.button_padx,
                               pady=self.button_pady,
                               command= lambda name='Debian1':self.eng.resume(name))
        self.buttonPs.pack(side=LEFT)
        self.buttonStop = Button(self.buttonsFrame, text="stop",
809                              width=self.button_width, padx=self.button_padx,
                                 pady=self.button_pady, command=self.stopEngine)
        self.buttonStop.pack(side=LEFT)

    def showXML(self):
814     file = open(self.loadconf.confFile, 'r')
        self.consoleText.insert(END, '\n')
        for line in file:
            self.consoleText.insert(END, line)
        file.close()
```

tkSimpleDialog.py

```
from Tkinter import *
2   import tkMessageBox
    import os
    import logging
    import loadConf

7   loadconf = None #set by gui

    class Dialog(Toplevel):

        def __init__(self, parent, title = None):
12          Toplevel.__init__(self, parent)
            self.transient(parent)
            if title:
                self.title(title)
            self.parent = parent
17          self.result = None
            body = Frame(self)
            self.initial_focus = self.body(body)
            body.pack(padx=5, pady=5)
            self.buttonbox()
22          self.grab_set()
            if not self.initial_focus:
                self.initial_focus = self
            self.protocol("WM_DELETE_WINDOW", self.cancel)
            self.geometry("+%d+%d" % (parent.winfo_rootx()+50,
```

```
                                  parent.winfo_rooty()+50))
        self.initial_focus.focus_set()
        self.wait_window(self)

    def body(self, master):
        # create dialog body.  return widget that should have
        # initial focus.  this method should be overridden
        pass

    def buttonbox(self):
        box = Frame(self)
        w = Button(box, text="OK", width=10, command=self.ok, default=ACTIVE)
        w.pack(side=LEFT, padx=5, pady=5)
        w = Button(box, text="Cancel", width=10, command=self.cancel)
        w.pack(side=LEFT, padx=5, pady=5)
        self.bind("&lt;Return>", self.ok)
        self.bind("&lt;Escape>", self.cancel)
        box.pack()

    def ok(self, event=None):
        if not self.validate():
            self.initial_focus.focus_set() # put focus back
            return
        self.withdraw()
        self.update_idletasks()
        self.apply()
        self.cancel()

    def cancel(self, event=None):
        # put focus back to the parent window
        self.parent.focus_set()
        self.destroy()

    def validate(self):
        return 1 # override

    def apply(self):
        pass # override


class moduleDialog(Dialog):

    def body(self, master):
        Label(master, text="Module:").grid(row=0)
        Label(master, text="Class:").grid(row=1)
        self.e1 = Entry(master, bg='white')
        self.e2 = Entry(master, bg='white')
        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1, column=1)
        self.module = ''
        self.className = ''
        return self.e1 # initial focus

    def validate(self):
        try:
            self.module = self.e1.get()
            self.className = self.e2.get()
            return 1
        except ValueError:
            tkMessageBox.showwarning(
                "Bad input",
                "Illegal values, please try again"
                )
            return 0

    def apply(self):
        pass

class vmDialog(Dialog):
```

```
97      def body(self, master):
            Label(master, text="VM:").grid(row=0)
            Label(master, text="IP:").grid(row=1)
            Label(master, text="PORT:").grid(row=2)
            self.e1 = Entry(master, bg='white')
102         self.e2 = Entry(master, bg='white')
            self.e3 = Entry(master, bg='white')
            self.e1.grid(row=0, column=1)
            self.e2.grid(row=1, column=1)
            self.e3.grid(row=2, column=1)
107         self.vmName = ''
            self.vmIp = ''
            self.vmPort = ''
            return self.e1 # initial focus

112     def validate(self):
            #validare l'ip
            try:
                self.vmName = self.e1.get()
                self.vmIp = self.e2.get()
117             self.vmPort = self.e3.get()
                return 1
            except ValueError:
                tkMessageBox.showwarning(
                    "Bad input",
122                 "Illegal values, please try again"
                    )
                return 0

        def apply(self):
127         pass

    class actionDialog(Dialog):

        def __init__(self, parent, loggingLevels = None, title = None):
132         self.loggingLevels = loggingLevels
            Dialog.__init__(self, parent, title)
            #print loggingLevels

        def body(self, master):
137         self.width = 15
            self.action = ''
            self.level = ''
            self.module = ''
            Label(master, text="TYPE").grid(row=0, column=0, sticky=N)
142         Label(master, text="DEFAULT").grid(row=0, column=1, sticky=N)
            Label(master, text="CUSTOM").grid(row=0, column=2, sticky=N)
            frameRadio = Frame(master)
            frameRadio.grid(row=1,column=0, padx=15, pady=15, sticky=NW)
            frameDefault = Frame(master)
147         frameDefault.grid(row=1,column=1, padx=15, pady=15, sticky=NW)
            frameCustom = Frame(master)
            frameCustom.grid(row=1,column=2, padx=15, pady=15, sticky=NW)
            self.radio = IntVar()
            self.radioButton1 = Radiobutton(frameRadio, text="Default",
152                                         variable=self.radio, value=1).pack(anchor=NW, padx=5, pady=5)
            self.radioButton2 = Radiobutton(frameRadio, text="Custom",
                                            variable=self.radio, value=2).pack(anchor=NW,  padx=5, pady=5)
            #default
            Label(frameDefault, text="Level:").grid(row=0, column=0, sticky=NW, padx=5, pady=5)
157         Label(frameDefault, text="Action:").grid(row=1, column=0, sticky=NW, padx=5, pady=5)
            self.levelChosenDefault = StringVar()
            items = self.loggingLevels.items()
            self.levelChosenDefault.set(items[0][1])
            self.e1 = Menubutton(frameDefault, width=self.width,
162                              textvariable=self.levelChosenDefault,
                                 relief=RAISED)
            self.e1.grid(row=0,column=1, padx=5, pady=5, sticky=NW)
```

```
        self.e1.menu = Menu(self.e1, tearoff=0 )
        self.e1["menu"] = self.e1.menu
        values = self.loggingLevels.keys()
        values.sort()
        for value in values:
            if value < loadconf.applicationStartingLevel:
                self.e1.menu.add_radiobutton(
                    label='%s:%s'%(value, self.loggingLevels[value]),
                    variable=self.levelChosenDefault,
                    value=self.loggingLevels[value])
        self.actionChosenDefault = StringVar()
        self.actionChosenDefault.set(loadconf.actionNames[0])
        self.e2 = Menubutton(frameDefault, width=self.width,
                             textvariable=self.actionChosenDefault,
                             relief=RAISED)
        self.e2.grid(row=1,column=1, padx=5, pady=5, sticky=NW)
        self.e2.menu = Menu(self.e2, tearoff=0 )
        self.e2["menu"] = self.e2.menu
        for value in loadconf.actionNames:
            self.e2.menu.add_radiobutton(
                label='%s'%value,
                variable=self.actionChosenDefault,
                value=value)

        #custom
        Label(frameCustom, text="level:").grid(row=0, column=0, padx=5, pady=5, sticky=NW)
        Label(frameCustom, text="name:").grid(row=1, column=0, padx=5, pady=5, sticky=NW)
        Label(frameCustom, text="module:").grid(row=2, column=0, padx=5, pady=5, sticky=NW)
        self.levelChosenCustom = StringVar()
        items = self.loggingLevels.items()
        self.levelChosenCustom.set(items[0][1])
        self.e3 = Menubutton(frameCustom, width=self.width,
                             textvariable=self.levelChosenCustom,
                             relief=RAISED)
        self.e3.grid(row=0,column=1, padx=5, pady=5, sticky=NW)
        self.e3.menu = Menu(self.e3, tearoff=0 )
        self.e3["menu"] = self.e3.menu
        values = self.loggingLevels.keys()
        values.sort()
        for value in values:
            if value < loadconf.applicationStartingLevel:
                self.e3.menu.add_radiobutton(
                    label='%s:%s'%(value, self.loggingLevels[value]),
                    variable=self.levelChosenCustom,
                    value=self.loggingLevels[value])
        self.e4 = Entry(frameCustom, bg='white')
        self.e4.grid(row=1, column=1, padx=5, pady=5, sticky=NW)
        self.e5 = Entry(frameCustom, bg='white')
        self.e5.grid(row=2, column=1, padx=5, pady=5, sticky=NW)

        return self.radioButton1 # initial focus


    def validate(self):
        try:
            if self.radio.get() == 1:
                self.level = self.levelChosenDefault.get()
                self.action = self.actionChosenDefault.get()
            elif self.radio.get() == 2:
                self.level = self.levelChosenCustom.get()
                self.action = self.e4.get()
                self.module = self.e5.get()
            else:
                return 0
            return 1
        except ValueError:
            tkMessageBox.showwarning(
                "Bad input",
```

```
                "Illegal values, please try again"
                )
            return 0

    def apply(self):
        pass

class levelDialog(Dialog):

    def body(self, master):
        Label(master, text="Gravity:").grid(row=0)
        Label(master, text="Name:").grid(row=1)
        self.e1 = Entry(master, bg='white')
        self.e2 = Entry(master, bg='white')
        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1, column=1)
        self.levelGravity = ''
        self.levelName = ''
        return self.e1 # initial focus

    def validate(self):
        try:
            self.levelGravity = self.e1.get()
            self.levelName = self.e2.get()
            return 1
        except ValueError:
            #controllo validita' ip
            tkMessageBox.showwarning(
                "Bad input",
                "Illegal values, please try again"
                )
            return 0

    def apply(self):
        pass

class serverDialog(Dialog):

    def body(self, master):
        Label(master, text="Ip:").grid(row=0)
        Label(master, text="Port:").grid(row=1)
        self.e1 = Entry(master, bg='white')
        self.e2 = Entry(master, bg='white')
        self.e1.grid(row=0, column=1)
        self.e2.grid(row=1, column=1)
        self.serverIp = ''
        self.serverPort = ''
        return self.e1 # initial focus

    def validate(self):
        try:
            self.serverIp = self.e1.get()
            self.serverPort = self.e2.get()
            return 1
        except ValueError:
            #controllo validita' ip
            tkMessageBox.showwarning(
                "Bad input",
                "Illegal values, please try again"
                )
            return 0

    def apply(self):
        pass
```

engine.py

```
import socket
import director
import logging
import traceback
import logParser
from loadConf import splitChar
import threading
import loadConf
import actions
import engineActions

class Engine:

    def __init__(self, loadconf):
        self.loadconf = loadconf
        actions.loadconf = self.loadconf
        self.myName = self.__class__.__name__

    def start(self):
        #load levels and start the logger
        self.createLogger()
        #logging start
        self.log(loadConf.START, self.myName + ' started')
        #event is used for signaling new logs to the logParser
        self.event = threading.Event()
        self.dir = director.Director(self.loadconf, self.event)
        self.parser = logParser.LogParser(self.loadconf, self.event)
        #start director
        self.dir.start()
        #start log parser
        self.parser.start()
        #start modules
        self.loadModules()

    def loadModules(self):
        for mod in self.loadconf.modules:#mod = (fileName, className)
            code = 'import %s' % mod[0]
            exec code
            code = 'self.%s = %s.%s(self.loadconf, self.event) ' % (mod[0], mod[0], mod[1])
            exec code
            code = 'self.%s.start()' % mod[0]
        exec code

    def config(self):
        self.log(loadConf.START, 'Configuration reloaded')
        for name, host in self.loadconf.vm.items():
            try:
                print '%s %s %s' % (host[0], host[1], self.loadconf.confFile)
                engineActions.sendConf(host[0], host[1], self.loadconf.confFile)
            except:
                self.log(loadConf.APPLERROR, 'engineActions.sendConf %s to %s:%s\n%s'
                        % (self.loadconf.confFile, host[0], host[1], (traceback.format_exc())[:-1]))
                return
            else:
                pass

    def ps(self, name):
        if self.loadconf.vm.has_key(name):
            host = self.loadconf.vm[name]
            try:
                ps = engineActions.showProc(host[0], host[1])
            except:
                self.log(self.loadconf.IDSERROR, 'engineActions.showProc to %s:%s\n%s'
                        % (host[0], host[1], (traceback.format_exc())[:-1]))
            else:
                print ps
```

```
            else:
                print 'vm name not found'

        def stop(self):
            for name, host in self.loadconf.vm.items():
                try:
                    engineActions.closeApp(host[0], host[1])
                except:
                    self.log(loadConf.APPLERROR, 'engineActions.closeApp %s:%s\n%s'
                             % (host[0], host[1], (traceback.format_exc())[:-1]))
            self.dir.stop()
            self.parser.stop()
            self.dir.join()
            self.parser.join()
            for mod in self.loadconf.modules:#mod = (fileName, className)
                code = 'import %s' % mod[0]
                exec code
                code = 'self.%s.stop() ' % mod[0]
                exec code
                code = 'self.%s.join()' % mod[0]
            exec code
            self.log(loadConf.STOP, self.myName + ' stopped')

        def log(self, name, msg):
            level = self.loadconf.levels[name]
            self.appLogger.log(level, msg)
            self.consoleLogger.log(level, msg)

        def createLogger(self):
            self.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + self.myName)
            self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + self.myName)
            actions.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + actions.myName)
            actions.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + actions.myName)
            #ids logger
            file = open(self.loadconf.idsLogFile, 'w')
            file.close()
            self.idsLogger = logging.getLogger(self.loadconf.idsLogRootName)
            self.idsLogger.setLevel(1)
            formatter = logging.Formatter(
                '%(asctime)s '+splitChar+' %(name)s '+splitChar+' %(levelname)s '+splitChar+' %(message)s')
            lfh = logging.FileHandler(self.loadconf.idsLogFile)
            lfh.setLevel(self.loadconf.idsLogFileLevel)
            lfh.setFormatter(formatter)
            self.idsLogger.addHandler(lfh)
```

engineActions.py

```
import socket
import sys, traceback
import loadConf

def closeApp(host, port):
    """
    Send the host the close application command
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(loadConf.socketTimeout)
    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 'close'
            s.close()
        else:
            s.close()

    def sendConf(host, port, file):
        """
        Send the configuration file to the host
        """
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(loadConf.socketTimeout)
        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
            #lanciare custom exception
        s.close()

    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(loadConf.socketTimeout)
        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()
```

director.py

```
import cPickle
import logging
import logging.handlers
import SocketServer
import struct
import logServer
from threading import Thread
import loadConf

class Director(Thread):

    def __init__(self, loadconf, event):
        Thread.__init__(self)
        self.loadconf = loadconf
        self.myName = self.__class__.__name__
        self.logger = logging.getLogger(self.loadconf.idsLogRootName + '.' + self.myName )
        self.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + self.myName)
        self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + self.myName)
        self.log(loadConf.START, self.myName + " started")
        self.tcpserver = logServer.LogRecordSocketReceiver(host=self.loadconf.serverIp,
                                                           port=self.loadconf.serverPort)
        logServer.event = event

    def log(self, name, msg):
        level = self.loadconf.levels[name]
        self.appLogger.log(level, msg)
        self.consoleLogger.log(level, msg)

    def run(self):
        self.tcpserver.serve_until_stopped()

    def stop(self):
        self.tcpserver.stop()
        self.log(loadConf.STOP, self.myName + " stopped")
```

actions.py

```
import libvirt
import sys
import os
import time
import loadConf
import logging
```

```
from threading import *

global myName
myName = __name__
loadconf = None #set by engine


def splitDomainName(name):
    domainName = name.split('.')
    print domainName
    if len(domainName) == 2:  #logger from module: logger.module
        return None
    else:
        #domain name is: logger.director.vmname.agent
        domain = domainName[2]
        return domain

def consoleLog(kargs):
    msg = kargs['NAME'] + ' ' + kargs['LEVEL'] + ' ' + kargs['MSG']
    log(loadConf.CONSOLELOG, msg)

def log(name, msg):
    level = loadconf.levels[name]
    appLogger.log(level, msg)
    consoleLogger.log(level, msg)

def run(time, name, level, msg):
    kargs = {'TIME': time, 'NAME': name, 'LEVEL': level, 'MSG': msg}
    if not loadconf.levelsActions.has_key(level):
        log('APPLERROR', 'No action associated to level %s: %s %s' % (repr(level), name, msg))
    else:
        action = loadconf.levelsActions[level][0]
        type = loadconf.levelsActions[level][1]
        if type == 'default':
            if loadconf.actionsFunctions.has_key(action):
                    loadconf.actionsFunctions[action](kargs)
            else:
                log('APPLERROR', 'No default action for %s: %s, %s ' % (action, name, msg))
        elif type == 'custom':
            module = loadconf.levelsActions[level][2]
            code = 'import %s' % module
            exec code
            if loadconf.actionsFunctions.has_key(action):
                loadconf.actionsFunctions[action](kargs)
            else:
                log('APPLERROR', 'No custom action for %s: %s, %s ' % (action, name, msg))
        else:
            log('APPLERROR', 'Not a type for action for %s: %s, %s ' % (action, name, msg))

def connect(domainName):
    if not os.access("/proc/xen", os.R_OK):
        log('APPLERROR', 'System is not running a Xen kernel')
        return [None, None]

    #connect to the hypervisor
    try:
        conn = libvirt.openReadOnly(None)
    except:
        log('APPLERROR', 'Failed to open connection to the hypervisor')
        return None, None

    #get a domain from its name
    try:
        dom = conn.lookupByName(domainName)
    except:
        log('APPLERROR', 'Failed to find the domain %s' + domainName)
        return None, None
    if conn is not None and dom is not None:
        return conn, dom
```

```
        else:
            return None, None

def stop(kargs):
    time = kargs['TIME']
    domainName = kargs['NAME']
    level = kargs['LEVEL']
    msg = kargs['MSG']
    name = splitDomainName(domainName)
    conn, dom = connect(name)
    if dom is not None:
        #suspend and resume a domain
        log('ACTION', 'Suspending domain %s for 60 seconds: %s' % (name, msg))
        if dom.suspend() != 0:
            log('APPLERROR', 'Failed to suspend domain %s' % name)
            return
        time.sleep(5)
        dom.resume()
    else:
        log('APPLERROR', 'Failed to suspend domain %s:%s' % (name, msg))

def save(kargs):
    time = kargs['TIME']
    domainName = kargs['NAME']
    level = kargs['LEVEL']
    msg = kargs['MSG']
    name = splitDomainName(domainName)
    if name is None:
        return
    conn, dom = connect(name)
    imgFile = name + '.img'
    if dom is not None:
        #save a domain to a file
        if dom.save(imgFile) != 0:
            log('APPLERROR', 'Impossible to save domain %s to file %s:%s' %(name, imgFile, msg))
        else:
            log('ACTION', 'Saved domain %s to file %s' %(name, imgFile))

    else:
        log('APPLERROR', 'Impossible to save domain %s to file %s:%s' %(name, imgFile, msg))


#called directly from the gui
def resumeFromFile(time, name, level, msg, imgFile):
    conn, dom = connect(name)
    #restore a domain from a file
    if conn.restore(imgFile) != 0:
        log('APPLERROR', 'Impossible to restore domain %s from file %s' %(name, imgFile))
    else:
        log('ACTION', 'Restored domain %s from file %s' %(name, imgFile))
```

logServer.py

```
import cPickle
import logging
import logging.handlers
import SocketServer
import struct
from threading import *

class LogRecordStreamHandler(SocketServer.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """
```

```
        def handle(self):
            """
            Handle multiple requests - each expected to be a 4-byte length,
            followed by the LogRecord in pickle format. Logs the record
            according to whatever policy is configured locally.
            """
            while 1:
                chunk = self.connection.recv(4)
                if len(chunk) < 4:
                    break
                slen = struct.unpack(">L", chunk)[0]
                chunk = self.connection.recv(slen)
                while len(chunk) < slen:
                    chunk = chunk + self.connection.recv(slen - len(chunk))
                obj = self.unPickle(chunk)
                record = logging.makeLogRecord(obj)
                self.handleLogRecord(record)

        def unPickle(self, data):
            return cPickle.loads(data)

        def handleLogRecord(self, record):
            # if a name is specified, we use the named logger rather than the one
            # implied by the record.
            if self.server.logname is not None:
                name = self.server.logname
            else:
                name = record.name
            logger = logging.getLogger(name) #name is engine.director.collector.agent

            # N.B. EVERY record gets logged. This is because Logger.handle
            # is normally called AFTER logger-level filtering. If you want
            # to do filtering, do it at the client end to save wasting
            # cycles and network bandwidth!
            logger.handle(record)
            event.set()

    class LogRecordSocketReceiver(SocketServer.ThreadingTCPServer):
        """simple TCP socket-based logging receiver suitable for testing.
        """

        allow_reuse_address = 1

        def __init__(self, host='localhost',
                     port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                     handler=LogRecordStreamHandler):
            SocketServer.ThreadingTCPServer.__init__(self, (host, port), handler)
            self.abort = 0
            self.timeout = 1
            self.logname = None

        def serve_until_stopped(self):
            import select
            sevent = event
            abort = 0
            while not abort:
                rd, wr, ex = select.select([self.socket.fileno()],
                                           [], [],
                                           self.timeout)
                if rd:
                    self.handle_request()

                abort = self.abort

        def stop(self):
            self.abort = 1
```

logParser.py

```
import time
import os
from director import *
from threading import *
import actions
import loadConf
from loadConf import splitChar
from thread import *

class LogParser(Thread):

    def __init__(self, loadconf, event):
        Thread.__init__(self)
        self.loadconf = loadconf
        self.event = event
        self.abort = 0
        self.myName = self.__class__.__name__
        self.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + self.myName)
        self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + self.myName)
        self.log(loadConf.START, self.myName + " started")
        self.interval = 5

    def log(self, name, msg):
        level = self.loadconf.levels[name]
        self.appLogger.log(level, msg)
        self.consoleLogger.log(level, msg)

    def run(self):
        input = open(self.loadconf.idsLogFile, 'r')
        old_len = len(input.readlines())
        input.close()
        abort = 0
        while not abort:
            self.event.wait(self.interval) #waiting for new logs
            self.event.clear()#woken up
            input = open(self.loadconf.idsLogFile, 'r')
            lines = input.readlines();
            new_len = len(lines)
            if(new_len != old_len):
                new_lines = lines[old_len-new_len:] #slice of the last lines
                for line in new_lines:
                    time, name, level, msg = line.split(splitChar, 3) #log msg is: time name level msg
                    start_new_thread(actions.run,(time.strip(), name.strip(), level.strip(), msg.strip()))
                old_len = new_len
            input.close()
            abort = self.abort
        self.log(loadConf.STOP, self.myName + " stopped")

    def stop(self):
        self.abort = 1
        self.event.set()
```

module.py

```
import logging
import director
from threading import Thread
import threading
import loadConf
import time
import sys
import loadConf

class Module(Thread):
```

```
12      def __init__(self, loadconf, event):
            Thread.__init__(self)
            #create logger
            self.myName = self.__class__.__name__
            self.loadconf = loadconf
17          self.logParserEvent = event
            self.idsLogger = logging.getLogger(self.loadconf.idsLogRootName + '.' + self.myName)
            self.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + self.myName)
            self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + self.myName)
            self.appLog(loadConf.START, self.myName + " started")
22          self._finished = threading.Event()
            self._interval = 100.0

        def setInterval(self, interval):
            self._interval = interval
27
        def idsLog(self, name, msg):
            level = self.loadconf.levels[name]
            self.idsLogger.log(level, msg)
            self.logParserEvent.set()
32
        def appLog(self, name, msg):
            level = self.loadconf.levels[name]
            self.appLogger.log(level, msg)
            self.consoleLogger.log(level, msg)
37
        def stop(self,timeout=None):
            self._finished.set()

        def task(self):
42          pass

        def run(self):
            while 1:
                if not self._finished.isSet():
47                  self.task()
                    self._finished.wait(self._interval)
                else:
                    self.appLog(loadConf.STOP, self.myName + " stopped")
                    return
```

moduleps.py

```
    import logging
    import director
    from threading import Thread
 4  import threading
    import loadConf
    import time
    import loadConf
    import os
 9  import module
    import engineActions
    import libvirtUtil
    import xenaccess

14  class Moduleps(module.Module):

        def __init__(self, loadconf, event):
            module.Module.__init__(self, loadconf, event)
            self.setInterval(10)
19          #self.xa = {}
            for hostname, (ip, port) in self.loadconf.vm.items():
                domId = libvirtUtil.getPidFromName(hostname.strip())
                #self.xa[domId] = xenaccess.Xenaccess(domId)
                #dictionary di oggetti xenaccess
```

```
24          self.xa = xenaccess.Xenaccess(domId)

        def task(self):
            for hostname, (ip, port) in self.loadconf.vm.items():
                domId = libvirtUtil.getPidFromName(hostname.strip())
29              ps = engineActions.showProc(ip, port)
                #print ps
                ps = ps[1:] #first line is PID
                ps2 = self.xa.processList()
                if len(ps) > (len(ps2) + 1): #al max un processo in piu': ps axu
34                  self.logger.error("error message")
```

modulehash.py

```
    import xenaccess
    import logging
 3  import director
    from threading import Thread
    import threading
    import loadConf
    import time
 8  import loadConf
    import os
    import module
    import libvirtUtil
    import hashlib
13
    def calculateHash(page):
        h = hashlib.sha512()
        h.update(page)
        return h.hexdigest()
18
    #checkhash pages, from a domId
    class Modulehash(module.Module):

        def __init__(self, loadconf, event):
23          module.Module.__init__(self, loadconf, event)
            self.loadconf = loadconf
            self.setInterval(200)
            self.firstTime = 1
            self.pages = {} #vmid, pid, pages
28          self.hashPages = {} #vmid, pid, hash
            self.xa = {}

            self.processlist = []
            self.processlist.append(1)
33
            for hostname, (ip, port) in self.loadconf.vm.items():
                #print hostname, ip, port
                vmId = libvirtUtil.getPidFromName(hostname)
                self.xa[vmId] = xenaccess.Xenaccess(vmId)
38              self.pages[vmId] = {}
                self.hashPages[vmId] = {}
                for pid in self.processlist:
                    self.pages[vmId][pid] = self.xa[vmId].processText(pid)
                    hashPages = []
43                  for page in self.pages[vmId][pid]:
                        if page is not None:
                            hashPages.append(calculateHash(page))
                        else:
                            hashPages.append(None)
48                  self.hashPages[vmId][pid] = hashPages


        def task(self):
            for hostname, (ip, port) in self.loadconf.vm.items():
53              vmId = libvirtUtil.getPidFromName(hostname)
```

```
            #controlla le pagine nuove
            for pid in self.processlist:
                pages = self.xa[vmId].processText(pid)
            for i in range(len(pages)):
58              #se la pagina salvata e' Null e quella nuova no la aggiungo
                if self.pages[vmId][pid][i] is None and pages[i] is not None:
                    self.pages[vmId][pid][i] = pages[i]
                #adesso check hash, se la pagina nuova e' non Null
                if pages[i] is not None:
63                  newHash = calculateHash(pages[i])
                    #se il vecchio hash e' vuoto aggiungo quello nuovo
                    if self.hashPages[vmId][pid][i] is None:
                        self.hashPages[vmId][pid][i] = newHash
                    #altrimenti, faccio il confronto con quello vecchio
68                  else:
                        if self.hashPages[vmId][pid][i] != newHash:
                            self.idsLog('ERROR', self.myName + " different hash")
                        else:
                            pass
```

collector.py

```
    #! /usr/bin/env python
    # Echo server program
 3  import logging, logging.handlers
    import socket
    import loadConf
    import os
    import collectorActions
 8  import agentSnort
    import agentLogin

    vmName = 'Debian2'

13  class Collector:
        def __init__(self):
            self.myName = self.__class__.__name__
            #self.loggingLevels = {}
            #self.defaultLoggingNames = {}
18          self.confFile = 'confhash.xml'
            self.loadconf = loadConf.LoadConf(confFile=self.confFile, isDomain0=0)
            self.loadConfiguration()
            self.idsLogger = logging.getLogger(self.loadconf.idsLogRootName + '.' + vmName)
            self.idsLogger.setLevel(1)
23          socketHandler = logging.handlers.SocketHandler(self.loadconf.serverIp, self.loadconf.serverPort)
            self.idsLogger.addHandler(socketHandler)

        def addLevelNames(self):
            logging._levelNames = {}
28          for a, b in self.loadconf.levels.items():
                logging._levelNames[a] = b

        def loadConfiguration(self):
            self.loadconf.resetToDefault()
33          self.loadconf.load()
            self.addLevelNames()

        def log(self, name, msg):
            level = self.loadconf.levels[name]
38          self.idsLogger.log(level, msg)

        def start(self):
            self.log('DEBUG', self.myName + ' started')

43          host = self.loadconf.vm[vmName][0]
            port = self.loadconf.vm[vmName][1]
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
            self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.sock.bind((host, port))
48          self.sock.listen(5)
            abort = 0

            self.agentSnort = agentSnort.AgentSnort(self.loadconf)
            self.agentSnort.start()
53          self.agentLogin = agentLogin.AgentLogin(self.loadconf)
            self.agentLogin.start()

            while not abort:
                conn, addr = self.sock.accept()
58              data = conn.recv(16)
                #if not data: break
                print data
                if data == 'CONFIG':
                    try:
63                      collectorActions.receiveConf(conn, self.loadconf.confFile)
                    except:
                        #do something
                        pass
                elif data == 'PS':
68                  try:
                        collectorActions.showProc(conn)
                    except:
                        #do something
                        pass
73              conn.close()
                elif data == 'CLOSEAPP':
                    try:
                        abort = 1;
                        collectorActions.closeApp(conn)
78                  except:
                        #do something
                        pass
                else:
                    #do something
83                  pass
            self.sock.close()
            self.agentSnort.stop()
            self.agentLogin.stop()
            self.log('DEBUG', self.myName + ' stopped')
88
    if __name__ == '__main__':
        c=Collector()
        c.start()
```

collectorActions.py

```
    import os
    import socket
    import sys, traceback
4
    def closeApp(conn):
        msg = 'OK'
        while (len(msg) > 0):
            try:
9               ns = conn.send(msg)
            except:
                conn.close()
                raise
            msg = msg[ns:]
14      data = conn.recv(16)
        if not data:
            #todo
            #print 'closeApp connection closed'
            conn.close()
```

```
19
    def receiveConf(conn, file):
        msg = 'OK'
        while (len(msg) > 0):
            try:
24              ns = conn.send(msg)
            except:
                conn.close()
                raise
            msg = msg[ns:]
29      try:
            socketFile = conn.makefile('r', 0)
            lines = socketFile.readlines()
        except:
            socketFile.close()
34          conn.close()
            raise
        output = open(file, 'w')
        output.writelines(lines)
        socketFile.close()
39      output.close()
        conn.close()

    def showProc(conn):
        command = "ps axu"
44      output = os.popen(command)
        lines = output.readlines()
        try:
            try:
                socketFile = conn.makefile('w', 0)
49              socketFile.writelines(lines)
            except:
                raise
        finally:
            socketFile.close()
54          #deve chiudere prima l'altro lato
            conn.close()
```

agentLogin.py

```
    import logging
    import time
    import os
    from threading import *
 5  import loadConf
    from loadConf import splitChar
    from thread import *

    vmName = 'Debian2'
10  class AgentLogin(Thread):

        def __init__(self, loadconf):
            Thread.__init__(self)
            self.loadconf = loadconf
15          self.abort = 0
            self.myName = self.__class__.__name__
            self.idsLogger = logging.getLogger(self.loadconf.idsLogRootName + '.' + vmName + '.' + self.myName)
            self.log('DEBUG', self.myName + ' started')
            self._finished = Event()
20          self._interval = 1.0


        def log(self, name, msg):
            level = self.loadconf.levels[name]
25          self.idsLogger.log(level, msg)

        def run(self):
```

```
            fileName = '/var/log/auth.log'
            input = open(fileName, 'r')
            old_len = len(input.readlines())
            input.close()
            abort = 0
            while not abort:
                input = open(fileName, 'r')
                lines = input.readlines()
                new_len = len(lines)
                if(new_len != old_len):
                    new_lines = lines[old_len-new_len:] #slice of the last lines
                    for line in new_lines:
                        print line
                        line = line.replace('\n','')
                        if (line.find('FAILED LOGIN') != -1) or (line.find('Failed password') != -1)\
                        or (line.find('authentication failure') != -1):
                            self.log('ERROR', line)
                    old_len = new_len
                input.close()
                self._finished.wait(self._interval)
                abort = self.abort
            self.log(loadConf.STOP, self.myName + " stopped")

        def stop(self):
            self.abort = 1
            self._finished.set()
```

agentSnort.py

```
    import logging
    import time
    import os
    from threading import *
    import loadConf
    from loadConf import splitChar
    from thread import *

    vmName = 'Debian2'
    class AgentSnort(Thread):

        def __init__(self, loadconf):
            Thread.__init__(self)
            self.loadconf = loadconf
            self.abort = 0
            self.myName = self.__class__.__name__
            self.idsLogger = logging.getLogger(self.loadconf.idsLogRootName + '.' + vmName + '.' + self.myName)
            self.log('DEBUG', self.myName + ' started')
            self._finished = Event()
            self._interval = 1.0


        def log(self, name, msg):
            level = self.loadconf.levels[name]
            self.idsLogger.log(level, msg)

        def run(self):
            fileName = '/var/log/snort/alert'
            input = open(fileName, 'r')
            old_len = len(input.readlines())
            input.close()
            abort = 0
            while not abort:
                input = open(fileName, 'r')
                lines = input.readlines()
                new_len = len(lines)
                if(new_len != old_len):
                    new_lines = lines[old_len-new_len:] #slice of the last lines
```

```
                for line in new_lines:
                    print line
                    line = line.replace('\n','')
                    self.log('ERROR', line)
                old_len = new_len
            input.close()
            self._finished.wait(self._interval)
            abort = self.abort
        self.log(loadConf.STOP, self.myName + " stopped")

    def stop(self):
        self.abort = 1
        self._finished.set()
```

loadConf.py

```
from xml.sax import saxutils
from xml.sax import make_parser
from xml.sax.handler import feature_namespaces

global splitChar
global socketTimeout
splitChar = '$'
socketTimeout = 1
START='START'
STOP='STOP'
LOAD='LOAD'
ACTION='ACTION'
CONSOLELOG='CONSOLELOG'
APPLNOTSET='APPLNOTSET'
APPLDEBUG='APPLDEBUG'
APPLINFO='APPLINFO'
APPLWARNING='APPLWARING'
APPLERROR='APPLERROR'
APPLCRITICAL='APPLCRITICAL'

class LoadConf:
    def __init__(self, confFile='conf.xml', isDomain0='1'):
        self.confFile = confFile
        self.isDomain0 = isDomain0

    def resetToDefault(self):
        self.modules = []
        self.levelsActions = {} #associates level name with action name
        #associates action name with functions
        if self.isDomain0:
            import actions
            self.actionsFunctions = {'STOP' : actions.stop, 'CONSOLELOG' : actions.consoleLog,
                                     'SAVE' : actions.save}
            self.actionNames = ['STOP', 'CONSOLELOG', 'NONE', 'PAUSE', 'SAVE']
        #console logging
        self.consoleLogRootName = 'guilogger'
        self.consoleLevel = 100
        #ids logging
        self.idsLogFile = 'ids.log'
        self.idsLogRootName = 'idslogger'
        if not self.isDomain0:
            self.idsLogRootName += '.director'
        self.idsLogFileLevel = 0
        #application logging
        self.appLogFile = 'appl.log'
        self.appLogFileLevel = 100
        self.appLogRootName = 'applogger'
        self.serverDefaultIp = '0.0.0.0'
        self.serverDefaultPort = 7890
        self.clientDefaultPort = 5555
        self.serverIp = ''
```

```
52          self.serverPort = 0
            self.vm = {}
            self.applicationStartingLevel = 100
            self.levels = {START: 111,
                           LOAD : 112,
57                         STOP: 113,
                           ACTION : 115,
                           CONSOLELOG: 160,
                           APPLNOTSET: 100,
                           APPLDEBUG : 110,
62                         APPLINFO : 120,
                           APPLWARNING : 130,
                           APPLERROR: 140,
                           APPLCRITICAL : 150,
                           111 : START,
67                         112 : LOAD,
                           113 : STOP,
                           115 : ACTION,
                           160 : CONSOLELOG,
                           100 : APPLNOTSET,
72                         110 : APPLDEBUG,
                           120 : APPLINFO,
                           130 : APPLWARNING,
                           140 : APPLERROR,
                           150 : APPLCRITICAL
77                         }

        def setConfFile(self, confFile):
            self.confFile = confFile

82      def load(self):
            # Create a parser
            parser = make_parser()
            # Tell the parser we are not interested in XML namespaces
            parser.setFeature(feature_namespaces, 0)
87          # Create the handler
            dh = FindElement(self)
            # Tell the parser to use our handler
            parser.setContentHandler(dh)
            # Parse the input
92          parser.parse(self.confFile)

    def normSpace(text):
        "Remove redundant whitespace from a string"
        return ' '.join(text.split())
97
    class FindElement(saxutils.DefaultHandler):

        def __init__(self, loadConf):
            self.loadconf = loadConf
102         self.general = 0
            self.domain0 = 0
            self.domainU = 0
            self.hosts = 0
            self.modules = 0
107         self.actions = 0
            self.levels = 0
            self.logging = 0
            self.parameters = 0

112     def startElement(self, name, attrs):
            if self.general:
                if self.levels:
                    if name == 'level':
                        self.levelName = normSpace(attrs.get('name', ""))
117                     self.levelGravity = int(normSpace(attrs.get('gravity', "")))
                elif self.logging:
                    if name == 'logFile':
                        type = normSpace(attrs.get('type', ""))
```

```
                    if type == 'ids':
                        self.loadconf.idsLogFile = normSpace(attrs.get('name', ""))
                        self.loadconf.idsLogFileLevel = int(normSpace(attrs.get('level', "")))
                    elif type == 'application':
                        self.loadconf.appLogFile = normSpace(attrs.get('name', ""))
                        self.loadconf.appLogFileLevel = int(normSpace(attrs.get('level', "")))
                    elif type == 'console':
                        #here, only the level can be configured
                        self.loadconf.consoleLevel = int(normSpace(attrs.get('level', "")))
            elif name == 'levels':
                self.levels = 1
        elif self.domain0 and self.loadconf.isDomain0:
            if self.modules:
                if name == 'module':
                    self.moduleName = normSpace(attrs.get('name', ""))
                    self.moduleClass = normSpace(attrs.get('class', ""))
            elif self.actions:
                if name == 'action':
                    self.actionLevel = normSpace(attrs.get('level', ""))
                    self.actionExecute = normSpace(attrs.get('execute', ""))
                    self.actionType = normSpace(attrs.get('type', ""))
                    if self.actionType == 'custom':
                        self.actionModule = normSpace(attrs.get('module', ""))
            elif self.parameters:
                if name == 'socket':
                    socketTimeout = int(normSpace(attrs.get('timeout', "")))
            elif name == 'modules':
                self.modules = 1
            elif name == 'actions':
                self.actions = 1
            elif name == 'logging':
                self.logging = 1
            elif name == 'parameters':
                self.parameters=1
        elif self.domainU:
            if self.hosts:
                if name == 'server':
                    self.loadconf.serverIp = normSpace(attrs.get('ip', ""))
                    self.loadconf.serverPort = int(normSpace(attrs.get('port', "")))
                if name == 'vm':
                    self.vmName = normSpace(attrs.get('name', ""))
                    self.vmIp = normSpace(attrs.get('ip', ""))
                    self.vmPort = int(normSpace(attrs.get('port', "")))
            elif name == 'hosts':
                self.hosts = 1
        elif name == 'general':
            self.general = 1
        elif name == 'domain0':
            self.domain0 = 1
        elif name == 'domainU':
            self.domainU = 1

    def endElement(self, name):
        if self.general:
            if name == 'general':
                self.general = 0
            elif self.levels:
                if name == 'level':
                    self.loadconf.levels[self.levelName] = int(self.levelGravity)
                    self.loadconf.levels[int(self.levelGravity)] = self.levelName
                elif name == 'levels':
                    self.levels = 0
            elif self.logging:
                if name == 'logging':
                    self.logging = 0
        elif self.domain0 and self.loadconf.isDomain0:
            if name == 'domain0':
                self.domain0 = 0
            elif self.modules:
```

```
                if name == 'module':
                    self.loadconf.modules.append((self.moduleName, self.moduleClass))
                elif name == 'modules':
                    self.modules = 0
            elif self.actions:
                if name == 'action':
                    if self.actionType == 'default':
                        self.loadconf.levelsActions[self.actionLevel] = (self.actionExecute,
                                                                         self.actionType)
                    else:
                        self.loadconf.levelsActions[self.actionLevel] = (self.actionExecute,
                                                                         self.actionType, self.actionModule)
                        code = 'import %s' % self.actionModule
                        exec(code)
                        code = 'self.loadconf.actionsFunctions[self.actionExecute] = %s.%s'\
                               % (self.actionModule, self.actionExecute)
                        exec(code)
                elif name == 'actions':
                    self.actions = 0
            elif self.parameters:
                if name == 'parameters':
                    self.parameters = 0
        elif self.domainU:
            if name == 'domainU':
                self.domainU = 0
            elif self.hosts:
                if name == 'vm':
                    self.loadconf.vm[self.vmName] = (self.vmIp, self.vmPort)
                elif name == 'hosts':
                    self.hosts = 0
```

saveConf.py

```
import logging
import logging.handlers
import loadConf

class SaveConf:
    def __init__(self, loadconf, file, myLogging, modules, levels, actions, hosts, parameters):
        self.myName = self.__class__.__name__
        self.loadconf = loadconf
        self.appLogger = logging.getLogger(self.loadconf.appLogRootName + '.' + self.myName)
        self.consoleLogger = logging.getLogger(self.loadconf.consoleLogRootName + '.' + self.myName)
        self.ind = 0
        self.file = file
        self.myLogging = myLogging
        self.modules = modules
        self.levels = levels
        self.actions = actions
        self.hosts = hosts
        self.parameters = parameters

    def log(self, name, msg):
        level = self.loadconf.levels[name]
        self.appLogger.log(level, msg)
        self.consoleLogger.log(level, msg)

    def indent(self):
        curr='\t'*self.ind
        self.output.write(curr)

    def openTag(self, name):
        self.indent()
        self.output.write('<' + name + '>\n')

    def closeTag(self, name):
        self.indent()
```

```
        self.output.write('</' + name + '>\n')

    def writeValues(self, values):
        for tag, list in values.items():
            for elem in list:
                line = tag
                for attr, value in elem.items():
                    line+= ' %s="%s"' % (attr, value)
                self.openTag(line)
                self.closeTag(tag)

    def writeAll(self):
        self.output = open(self.file, 'w')
        self.openTag('configuration')
        self.ind+=1
        self.openTag('general')
        self.writeLevels()
        self.writeLogging()
        self.closeTag('general')
        self.openTag('domain0');
        self.writeModules()
        self.writeActions()
        self.writeParameters()
        self.closeTag('domain0')
        self.openTag('domainU')
        self.writeHosts()
        self.closeTag('domainU');
        self.ind-=1
        self.closeTag('configuration')
        self.output.close()
        self.log(loadConf.APPLDEBUG, 'Configuration saved on file %s' % self.file)

    def writeLogging(self):
        self.ind+=1
        self.openTag('logging')
        self.ind+=1
        self.writeValues(self.myLogging)
        self.ind-=1
        self.closeTag('logging')
        self.ind-=1

    def writeModules(self):
        self.ind+=1
        self.openTag('modules')
        self.ind+=1
        self.writeValues(self.modules)
        self.ind-=1
        self.closeTag('modules')
        self.ind-=1

    def writeLevels(self):
        self.ind+=1
        self.openTag('levels')
        self.ind+=1
        self.writeValues(self.levels)
        self.ind-=1
        self.closeTag('levels')
        self.ind-=1

    def writeActions(self):
        self.ind+=1
        self.openTag('actions')
        self.ind+=1
        self.writeValues(self.actions)
        self.ind-=1
        self.closeTag('actions')
        self.ind-=1

    def writeHosts(self):
```

```
            self.ind+=1
            self.openTag('hosts')
            self.ind+=1
            self.writeValues(self.hosts)
            self.ind-=1
            self.closeTag('hosts')
            self.ind-=1

        def writeParameters(self):
            self.ind+=1
            self.openTag('parameters')
            self.ind+=1
            self.writeValues(self.parameters)
            self.ind-=1
            self.closeTag('parameters')
            self.ind-=1
```

libvirtUtil.py

```
import libvirt
import sys

def getPidFromName(name):
    conn = libvirt.openReadOnly(None)
    if conn == None:
        print 'Failed to open connection to the hypervisor'
        return None
        #sys.exit(1)
    try:
        dom = conn.lookupByName(name)
    except:
        print 'Failed to find the main domain'
        return None
        #sys.exit(1)
    id = dom.ID()
    del dom
    del conn
    return id
```

xenaccess.py

```
#load library and define return types and argument types
from ctypes import *

xenaccess = cdll.LoadLibrary('./libxenaccess.so')
#char **process_list(uint32_t dom, int *proc_num)
xenaccess.process_list.restype = POINTER(c_char_p)
xenaccess.process_list.argtypes = [c_int, POINTER(c_int)]
#unsigned char **process_text(uint32_t dom, int pid, int *pages, int *page_size, uint32_t *last_offset))
xenaccess.process_text.restype = POINTER(POINTER(c_ubyte))
xenaccess.process_text.argtypes = [c_int, c_int, POINTER(c_int), POINTER(c_int), POINTER(c_int)]

class Xenaccess:

    def __init__(self, domId):
        self.domId = domId

    def processText(self, pid):
        numPages = c_int()
        pageSize = c_int()
        offset = c_int()
        ret = xenaccess.process_text(self.domId, c_int(pid), byref(numPages), byref(pageSize), byref(offset))
        if bool(ret) == False:
            return None
```

```
        else:
            text=[]
            for i in range(numPages.value):
                s=''
                if bool(ret[i]) == False:
                    text.append(None)
                else:
                    if(i == (numPages.value-1)):
                        for j in range(offset.value):
                            s=s+'%c'%ret[i][j]
                    else:
                        for j in range(pageSize.value):
                            if ret[i] != None:
                                s=s+'%c'%ret[i][j]
                    text.append(s)
            return text

    def processList(self):
        numProc = c_int()
        ret = xenaccess.process_list(self.domId, byref(numProc))
        if ret is None:
            return None
        else:
            ps=[]
            for i in range(numProc.value):
                ps.append(ret[i])
            del ret
            return ps
```

xenaccess_functions.c

```
#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/mman.h>
#include <stdio.h>
#include "xenaccess.h"
#include "xa_private.h"

#include <libvirt/libvirt.h>

#define TASKS_OFFSET 24 * 4   /* task_struct->tasks */
#define PID_OFFSET 39 * 4     /* task_struct->pid */
#define NAME_OFFSET 108 * 4   /* task_struct->comm */


char **process_list(uint32_t dom, int *proc_num)
{
    xa_instance_t xai;
    unsigned char *memory = NULL;
    uint32_t offset, next_process, list_head;
    char *name = NULL;
    int pid = 0;
    char **ret = NULL;
    char *info = NULL;
    int num_proc = 0;

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

    memory = xa_access_kernel_symbol(&xai, "init_task", &offset);
    if (NULL == memory){
```

```
            perror("failed to get process list head");
            goto error_exit;
        }
        memcpy(&next_process, memory + offset + TASKS_OFFSET, 4);
        list_head = next_process;
        munmap(memory, XA_PAGE_SIZE);

        while (1){
            memory = xa_access_virtual_address(&xai, next_process, &offset);
            if (NULL == memory){
                perror("failed to map memory for process list pointer");
                goto error_exit;
            }
            memcpy(&next_process, memory + offset, 4);
            if (list_head == next_process){
                break;
            }

            name = (char *) (memory + offset + NAME_OFFSET - TASKS_OFFSET);
            memcpy(&pid, memory + offset + PID_OFFSET - TASKS_OFFSET, 4);

            info = (char *)malloc(sizeof(char) * (strlen(name) + 20));
            sprintf(info, "%d,%s", pid, name);

            if(num_proc == 0)
            {
                 ret = (char **)malloc(sizeof(char *));
                 ret[num_proc] = info;
                 num_proc++;
            }
            else
            {
              ret = realloc(ret, (num_proc+1)*(sizeof(char *)));
                 if(ret == NULL)
                 {
                      perror("failed to realloc memory");
                      goto error_exit;
                 }
                 ret[num_proc] = info;
                 num_proc++;

            }
            munmap(memory, XA_PAGE_SIZE);
            memory=NULL;
        }

    error_exit:

        xa_destroy(&xai);
        if (memory) munmap(memory, XA_PAGE_SIZE);
        *proc_num = num_proc;
        return ret;
    }


    unsigned char **process_text(uint32_t dom, int pid, int *pages, int *page_size, uint32_t *last_offset)
    {
        xa_instance_t xai;
        xa_linux_taskaddr_t taskaddr;
        unsigned char *memory = NULL;
        uint32_t offset = 0;
        unsigned long vaddr=0;
        int num_pages = 0;
        int page = 0;
        unsigned char *info;
        unsigned char **ret=NULL;
        unsigned long diff=0;
        unsigned long i=0;
```

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

    if (xa_linux_get_taskaddr(&xai, pid, &taskaddr) == XA_FAILURE){
        perror("failed to get task addresses");
113     goto error_exit;
    }

    num_pages = 1 + (int)((taskaddr.end_code - taskaddr.start_code) / (unsigned long)XC_PAGE_SIZE);
    ret = (unsigned char **)malloc(num_pages*(sizeof(unsigned char *)));
118
    for(vaddr = taskaddr.start_code; vaddr <= taskaddr.end_code; vaddr+=XC_PAGE_SIZE)
    {
         memory = xa_access_user_virtual_address(&xai, vaddr, &offset, pid);
         if (NULL == memory)
123      {
             ret[page] = NULL;
         }
         else
         {
128          info =(unsigned char*) malloc(XC_PAGE_SIZE * sizeof(unsigned char));
             memcpy(info, memory, XC_PAGE_SIZE);
             ret[page] = info;
             munmap(memory, XA_PAGE_SIZE);
             memory=NULL;
133      }
         page++;
    }

    if(ret[num_pages-1] != NULL)
138 {
         diff=0;
         diff=(XC_PAGE_SIZE*num_pages)-(taskaddr.end_code-taskaddr.start_code);
         for(i=1; i<=diff; i++)
         {
143          ret[num_pages-1][XC_PAGE_SIZE-i] = '\0';
         }
    }

error_exit:
148 xa_destroy(&xai);
    if (memory) munmap(memory, XC_PAGE_SIZE);
    *page_size = XC_PAGE_SIZE;
    *pages = num_pages;
    *last_offset = offset;
153 return ret;
}
```

Bibliografia

[1] AMD's Virtualization Solutions. http://enterprise.amd.com/us-en/Solutions/Consolidation/virtualization.aspx.

[2] The Apache Software Foundation. http://www.apache.org/.

[3] Bochs: The Open Source IA-32 Emulator. http://bochs.sourceforge.net/.

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

[5] Common Criteria. http://www.commoncriteriaportal.org.

[6] The ctypes package. http://starship.python.net/crew/theller/ctypes/.

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

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

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

[10] Intel Virtualizazion Technology. http://www.intel.com/technology/computing/vptech/.

[11] LaGrande Technology. http://www.intel.com/technology/security/.

[12] Libvirt: the virtualization api. http://libvirt.org/.

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

[14] Microsoft Virtual PC. http://www.microsoft.com/windows/virtualpc/default.mspx.

[15] Nessus Vulnerability Scanner. http://www.nessus.org/.

[16] NetTop. http://www.nsa.gov/techtrans/techt00011.cfm.

[17] nmap - free security scanner for network exploration and security audits. http://insecure.org/nmap/.

[18] NSA/NCSC Rainbow Series: NCSC-TG-004 [Aqua Book]. Glossary of computer security terms. http://www.fas.org/irp/nsa/rainbow/tg004.htm.

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

[20] QEMU: open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

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

[22] Snort - the de facto standard for intrusion detection/prevention. http://www.snort.org/.

[23] TkInter. http://wiki.python.org/moin/TkInter.

[24] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/.

[25] Virtuozzo. http://www.swsoft.com/en/products/virtuozzo/.

[26] VMware ESX server. http://www.vmware.com/products/vi/esx/.

[27] VMware Workstation. http://www.vmware.com/products/ws/.

[28] The Xen virtual machine monitor. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

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

[30] ANDERSON, J. P. Computer Security Technology Planning Study. Tech. Rep. ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[31] ANDRESEN, R. J. Virtual machine monitors. CERN OpenLab for Data grid application, August 2004.

[32] AXELSSON, S. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security* (New York, NY, USA, 1999), ACM Press, pp. 1–7.

[33] BAIARDI, F., AND TELMON, C. Detecting intruders by checking OS calls. In *Tiwis2003 Workshop* (May 2003).

[34] BALASUBRAMANIYAN, J. S., GARCIA-FERNANDEZ, J. O., ISACOFF, D., SPAFFORD, E. H., AND ZAMBONI, D. An architecture for intrusion detection using autonomous agents. In *ACSAC* (1998), pp. 13–24.

[35] BEN-YEHUDA, M., MASON, J. D., KRIEGER, O., AND XENIDIS, J. Xen/IOMMU: Breaking IO in new and interesting ways. `http://www.xensource.com/files/xs0106_xen_iommu.pdf`.

[36] BEN-YEHUDA, M., MASON, J. D., KRIEGER, O., XENIDIS, J., DORN, L. V., MALLICK, A., NAKAJIMA, J., AND WAHLIG, E. Using IOMMUs for virtualization in Linux and Xen. `http://www.mulix.org/lectures/using-iommus-for-virtualization/OLS-jdmason.pdf`.

[37] BISHOP, M. *Computer Security: Art and Science*. Addison-Wesley, 2003.

[38] BREWER, D. F. C., AND NASH, M. J. The chinese wall security policy. In *IEEE Symposium on Security and Privacy* (1989), pp. 206–214.

[39] DENNING, D. E. An intrusion-detection model. *IEEE Trans. Softw. Eng. 13*, 2 (1987), 222–232.

[40] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[41] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 211–224.

[42] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (October 2004).

[43] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)* (October 2003).

[44] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).

[45] GARFINKEL, T., AND ROSENBLUM, M. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (May 2005).

[46] GOLDBERG, R. P. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems* (New York, NY, USA, 1973), ACM Press, pp. 74–112.

[47] GOLDBERG, R. P. Survey of virtual machine research. *IEEE Computer 7*, 6 (1974), 34–45.

[48] HAND, S. Xen Security. `http://www.xensource.com/files/XenSecurity_SHand.pdf`.

[49] HEISER, G., ULHIG, V., AND LEVASSEUR, J. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev. 40*, 1 (2006), 95–99.

[50] HERVE DEBAR, DAVID A. CURRY, B. S. F. The intrusion detection message exchange format, March 2006. `http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-16.txt`.

[51] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of Computer Security 6*, 3 (1998), 151–180.

[52] HUAGANG, X. Build a secure system with LIDS. `http://www.lids.org/document/build_lids-0.2.html`.

[53] HUAGANG, X. LIDS Hacking HOWTO. `http://www.lids.org/lids-howto/lids-hacking-howto.html`.

[54] ILGUN, K., KEMMERER, R. A., AND PORRAS, P. A. State transition analysis: A rule-based intrusion detection approach. *Software Engineering 21*, 3 (1995), 181–199.

[55] JIANG, X., AND XU, D. Collapsar: A VM-based architecture for network attack detention center. In *USENIX Security Symposium* (2004), pp. 15–28.

[56] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 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, pp. 91–104.

[57] JR., N. L. P., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium* (2004), pp. 179–194.

[58] KAMP, P. H., AND WATSON, R. N. M. Jails: Confining the omnipotent root. In *In Proceedings of the 2nd International SANE Conference* (2000).

[59] KENT, K., AND MELL, P. Guide to Intrusion Detection and Prevention (IDP) systems (DRAFT). *National Institute of Standards and Technology* (2006).

[60] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. SubVirt: Implementing malware with virtual machines. *sp 0* (2006), 314–327.

[61] KOURAI, K., AND CHIBA, S. HyperSpector: virtual distributed monitoring environments for secure intrusion detection. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (New York, NY, USA, 2005), ACM Press, pp. 197–207.

[62] LEUNG, F., NEIGER, G., RODGERS, D., SANTONI, A., AND UHLIG, R. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal* (August 2006).

[63] LUTZ, M., AND ASCHER, D. *Learning Python, Second Edition*. O'Reilly, 2003.

[64] MCCUNE, J., BERGER, S., CACERES, R., JAEGER, T., AND SAILER, R. DeuTeRiuM: A system for distributed mandatory access control. IBM Research Report, February 2006.

[65] MILLER, T. Analysis of the t0rn rootkit. `http://www.securityfocus.com/infocus/1230`.

[66] PADEGS, A. System/370 extended architecture: Design considerations. *IBM Journal of Research and Development 27*, 3 (1983), 198–205.

[67] RASH, M., OREBAUGH, A. D., CLARK, G., PINKARD, B., AND BABBIN, J. *Intrusion Prevention and Active Response: Deploying Network and Host IPS*. Syngress Publishing, 2005.

[68] ROBIN, J., AND IRVINE, C. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor, 2000.

[69] ROSENBLUM, M. The reincarnation of virtual machines. *Queue 2*, 5 (2004), 34–40.

[70] ROSENBLUM, M., AND GARFINKEL, T. Virtual machine monitors: Current technology and future trends. *Computer 38*, 5 (2005), 39–47.

[71] SAILER, R., VALDEZ, E., JAEGER, T., PEREZ, R., VAN DOORN, L., GRIFFIN, J. L., AND BERGER, S. sHype: A secure hypervisor approach to trusted virtualized systems. IBM Research Report, 2005.

[72] SEAWRIGHT, L. H., AND MACKINNON, R. A. VM/370 - a study of multiplicity and usefulness. *IBM Systems Journal 18*, 1 (1979), 4–17.

[73] SEKAR, R., BOWEN, T. F., AND SEGAL, M. E. On preventing intrusions by process behavior monitoring. In *Workshop on Intrusion Detection and Network Monitoring* (1999), pp. 29–40.

[74] SINGH, A. An introduction to virtualization. `http://www.kernelthread.com/publications/virtualization/`.

[75] SNAPP, S. R., BRENTANO, J., DIAS, G. V., GOAN, T. L., HEBERLEIN, L. T., LIN HO, C., LEVITT, K. N., MUKHERJEE, B., SMAHA, S. E., GRANCE, T., TEAL, D. M., AND MANSUR, D. DIDS (Distributed Intrusion Detection System) - motivation, architecture, and an early prototype. In *Proceedings of the 14th National Computer Security Conference* (Washington, DC, 1991), pp. 167–176.

[76] SPARKS, S., AND BUTLER, J. Shadow Walker: raising the bar for rootkit detection. `hwww.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf`.

[77] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A., MARTING, F., ANDERSON, A., BENNETT, S., KAGI, A., LEUNG, F., AND SMITH, L. Intel Virtualization Technology. *Computer 38*, 5 (May 2005), 48–56.

[78] UNIVERSITY OF CAMBRIDGE UK. Xen interface manual: Xen v3.0 for x86. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface/interface.html`.

[79] WAHLIG, E., AND S.MCDOWELL. AMD Pacifica Virtualization Technology. http://www.xensource.com/files/Xen_PacificaDisclosure_AMD_EWahlig.pdf.

[80] WHITAKER, A., SHAW, M., AND GRIBBLE, S. Scale and performance in the Denali isolation kernel, 2002.

[81] Y.DONG, LI, S., MALLICK, A., NAKAJIMA, J., TIAN, K., XU, X., YANG, F., AND YU, W. Extending Xen* with Intel Virtualization Technology. *Intel Technology Journal* (August 2006).

[82] ZHANG, Y., AND PAXSON, V. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium* (August 2000), pp. 67–81.