UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA MAGISTRALE IN TECNOLOGIE
DELL'INFORMAZIONE E DELLA COMUNICAZIONE

ANALISI ED IMPLEMENTAZIONE DI UN ANOMALY HOST BASED INTRUSION DETECTION SYSTEM

Relatore: Prof. D. Bruschi
Correlatore: Dott. A. Lanzi
Correlatore: Dott. L. Cavallaro

Tesi di Laurea di
Emanuele Passerini
matr. 685431

Anno Accademico 2005/2006

Ai miei genitori che mi hanno dato tutto

RINGRAZIAMENTI

Ringrazio innanzitutto i miei genitori che mi hanno sostenuto e mi sono stati vicini in questi anni di studio, tutti gli amici con cui ho condiviso difficoltà e gioie e con cui ho passato momenti indimenticabili.

Ringrazio il professor Danilo Bruschi che mi ha fatto crescere la passione per la sicurezza informatica e mi ha consentito di scegliere questo argomento di tesi. Un ringraziamento particolare va all'Ing. Mattia Monga, ai dottorandi ma soprattutto amici Andrea Lanzi, Lorenzo Cavallaro e Lorenzo Martignoni, al laureando (come me) Gianpaolo Fresi Roglia e a tutti i compagni del Laboratorio LaSeR: senza i vostri consigli e il vostro supporto non ce l'avrei sicuramente fatta.

Un ringraziamento speciale è diretto ai miei amici e compagni di studio Enzo e Cesare, con i quali ho condiviso gioie e dolori della Specialistica. Voglio ringraziare anche Mauro, Santino ed Enrico per la compagnia nelle pause pranzo e caffè.

Non potrei certo dimenticare i miei amici di "casa" che mi sono sempre stati vicini e con cui ho passato la maggior parte del mio tempo libero: Bac, Edo, Kurz, Ogni, Set e Vic. Grazie Vale, perché mi hai sopportato in questi mesi, perché mi dai sempre conforto e credi in me, e perché mi rendi felice.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

I computer e le reti al giorno d'oggi dominano gran parte delle nostre vite. Tutti noi confidiamo in queste tecnologie ogni giorno quando lavoriamo, ci divertiamo, ci muoviamo per strada, comunichiamo. Il grande panico associato all'anno 2000 e agli attacchi dell'11 Settembre e le conseguenti paure legate alle infrastrutture dimostrano quanto siamo dipendenti da componenti in ultima analisi controllate da computer.
Anche se ci fidiamo di questi sistemi siamo coscienti della possibile esistenza di imperfezioni che possono essere sfruttate per fini criminosi. Tutto ciò ha creato il bisogno di una tecnologia di sicurezza che sia in grado di monitorare i sistemi e di individuare i tentativi di intrusione e gli attacchi. Questo sistema viene chiamato intrusion detection (IDS) ed è indipendente dagli altri sistemi o architetture di sicurezza. Nel campo dell'informatica la sicurezza informatica e il rilevamento di intrusioni hanno così assunto un ruolo fondamentale.
In questa tesi viene proposto lo studio e l'analisi delle proprietà di sicurezza di un modello di tipo anomaly detection rappresentante lo stato dell'arte degli Host Intrusion Detection System (HIDS).
Un HIDS è un meccanismo utilizzato per rilevare intrusioni all'interno di un sistema informatico.
Esistono principalmente due tipologie di tali sistemi: misuse detection e anomaly detection. La prima basa l'efficacia del rilevamento dell'attacco sulle signature, sequenze di byte (pattern) che caratterizzano, più o meno precisamente, codice maligno. La seconda necessita di una fase di learning in cui vengono raccolte e registrate le infor-

mazioni del comportamento normale di un'applicazione, e una fase di detection in cui ogni deviazione dal comportamento normale imparato che va oltre una soglia predefinita viene identificata come un attacco.
Diversi modelli di anomaly-based HIDS sono stati proposti: da quelli meno precisi che caratterizzano il comportamento di un programma basandosi solo sulle sequenze di chiamate di sistema (system call o syscall) eseguite, fino ad arrivare al *Virtual Path*, uno tra i migliori modelli esistenti in termini di precisione di detection fino ad ora sviluppata.
Tale modello è un'evoluzione dei modelli basati sulle syscall ed è innovativo in quanto utilizza lo stack come sorgente di informazioni. L'idea alla base del modello è quella di estrarre informazioni dallo stack e nella fase di learning generare un percorso astratto di esecuzione tra ogni coppia di syscall eseguite dal programma, chiamato Virtual Path, e salvarlo in una forma compatta in un database. In una successiva fase di detection si deciderà in real-time se ogni Virtual Path è valido cercando deviazioni da quelli che sono stati acquisiti in fase di learning. Tale modello teorico è in grado di rilevare alcuni attacchi che non possono essere rilevati da precedenti approcci.
Durante il lavoro di tesi sono state fornite soluzioni implementative non trattate nel modello teorico e tralasciate in letteratura ma che risultano determinanti nella valutazione dell'applicabilità del modello nei sistemi reali. In particolare si è fornito un metodo per la gestione della parallelizzazione dei processi e la sua naturale estensione ai thread, ed una soluzione per la gestione dei segnali, eventi asincroni che possono modificare le tracce e generare falsi allarmi.
Queste soluzioni hanno consentito di implementare un modello completo e funzionante su piattaforma GNU/Linux. Sono stati creati un tracer che nella fase di learning registra in un database le tracce del comportamento dell'applicazione ed un detector che nella fase di detection rileva in real-time le deviazioni rispetto alle tracce generando allarmi.
Il prototipo ha permesso la verifica dei risultati teorici rispetto alla resistenza agli attacchi e all'efficienza di tale modello.

Capitolo 1

Sicurezza dell'informazione

1.1 Impatti dell'hacking

Un sistema informatico dovrebbe fornire confidenzialità, integrità e disponibilità dei dati. Con il termine confidenzialità si intende la protezione dei dati e delle informazioni scambiate tra un mittente e uno o più destinatari nei confronti di terze parti. In un sistema che garantisce la confidenzialità, una terza parte che entri in possesso delle informazioni scambiate tra mittente e destinatario, non è in grado di ricavarne alcun contenuto informativo intelligibile. Con il termine integrità si intende la protezione dei dati e delle informazioni nei confronti delle modifiche del contenuto effettuate da una terza parte, essendo compreso nell'alterazione anche il caso limite della generazione ex novo di dati ed informazioni. Insito nel concetto di integrità vi è la possibilità di verificare con assoluta certezza se un dato o una informazione sono rimasti integri, ossia inalterati nel loro contenuto, durante la loro trasmissione e/o la loro memorizzazione. In un sistema che garantisce l'integrità, l'azione di una una terza parte di modifica del contenuto delle informazioni scambiate tra mittente e destinatario, viene quindi rilevata. Infine con il termine disponibilità si intende la prevenzione della non accessibilità, ai legittimi utilizzatori, sia delle informazioni che delle risorse, quando informazioni e risorse servono. Il concetto quindi, oltre che riguardare dati ed informazioni, è esteso a tutte le possibili risorse che costituiscono un sistema informatico, come, ad esempio, la banda di trasmissione di un collegamento, la capacità di calcolo di un elaboratore, lo

spazio utile di memorizzazione dati, ecc. Tuttavia, vista la crescente diffusione della connettività ad Internet e al vasto spettro di possibilità di commercio e di operazioni finanziarie on-line, sempre più sistemi sono vittime di attacchi e tentativi di intrusione. Questi tentativi di eversione cercano di sfruttare debolezze o difetti dei sistemi operativi e delle applicazioni con risultati talvolta clamorosi come successo nel 1998 per l'Internet Worm.

Come riportato dal Computer Emergency Response Team/Coordination Center (CERT/CC), il numero di attacchi a sistemi informatici ha subito una crescita esponenziale negli ultimi anni (Figura 1.1). Inoltre, la complessità e la precisione degli attacchi stanno

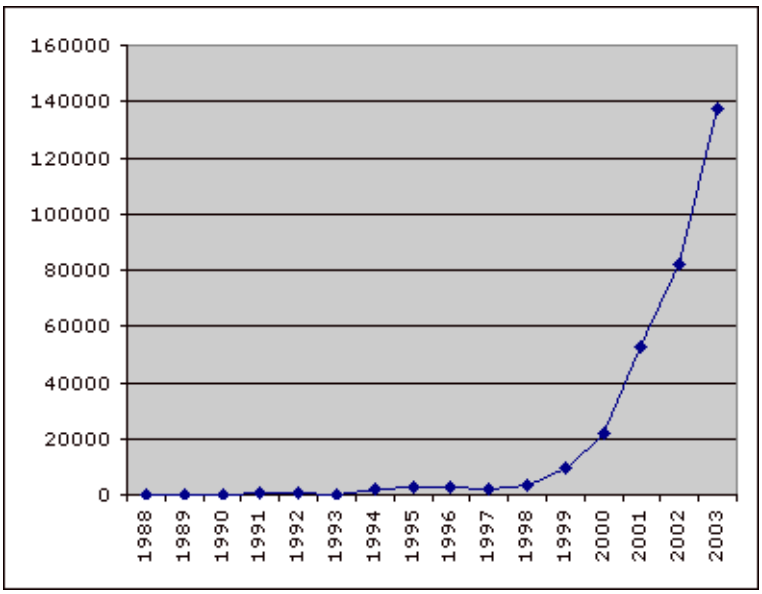

Figura 1.1: Crescita degli incidenti informatici riportati dal Computer Emergency Response Team/Coordination Center (CERT/CC)

anch'esse aumentando. Alcuni anni fa chi tentava di violare sistemi informatici aveva una profonda conoscenza informatica dei sistemi e delle reti che gli permettevano di effettuare attacchi; oggi invece chiunque potrebbe sfruttare una vulnerabilità di un sistema o di una rete utilizzando uno dei numerosi tool facilmente disponibili nella rete Internet.

Oggigiorno le perdite finanziarie dovute a virus risultano altissime: l'82% dei manager intervistati al CSI/FBI 2003 Eighth Annual Security Survey affermano che nell'ultimo

anno i virus sono diventati il loro più grande problema, e risulta inoltre che il 99% utilizzi software antivirus, che il 47% ha riportato perdite per più di 27 milioni di dollari. I virus e i worm rappresentano infatti una tremenda minaccia per la sicurezza delle organizzazioni e delle aziende nonostante queste si organizzino e proteggano in diversi modi. In pubblicazioni e articoli recenti si associa lo stato di sicurezza di un sistema (applicazione, sistema operativo, server..) alla proliferazione di virus e worm, generalmente dovute a vulnerabilità e bug, sempre maggiori in numero e complessità. Per questi motivi la gestione degli aggiornamenti e delle patch e l'"hardenizzazione" dei sistemi stanno diventando un passo fondamentale nella strategia da adottare per mettere in sicurezza un sistema informatico.

L'abilità di gestire le vulnerabilità di un sistema e ridurre l'esposizione agli attacchi è ormai fondamentale per la sopravvivenza di ogni organizzazione e azienda. Come dimostrato dalla Figura 1.2, presa dal rapporto annuale della Symantec [1], il numero e la pericolosità delle vulnerabilità sta aumentando e questa è una chiara dimostrazione di quanto numerose siano le minacce che minano la sicurezza delle organizzazioni e quanto grande sia la necessità di perseguire la sicurezza.

Dalla Figura 1.3 è facile capire quante siano le nuove vulnerabilità ogni mese. Per

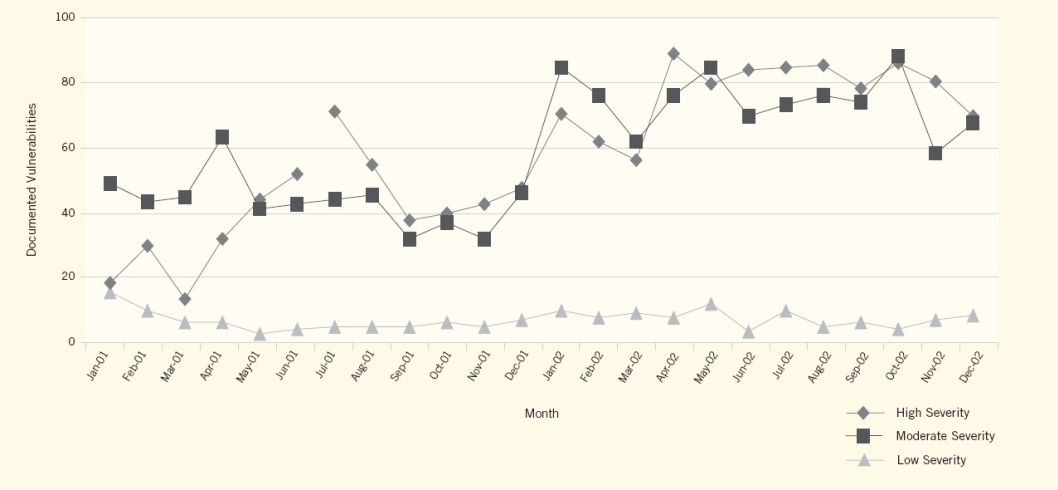

Figura 1.2: Aumento del volume delle vulnerabilità (Symantec, 2003)

le aziende coinvolte mantenere i propri sistemi sicuri significa inoltre raccogliere più informazioni possibili circa le nuove vulnerabilità e calcolarne l'eventuale impatto sui

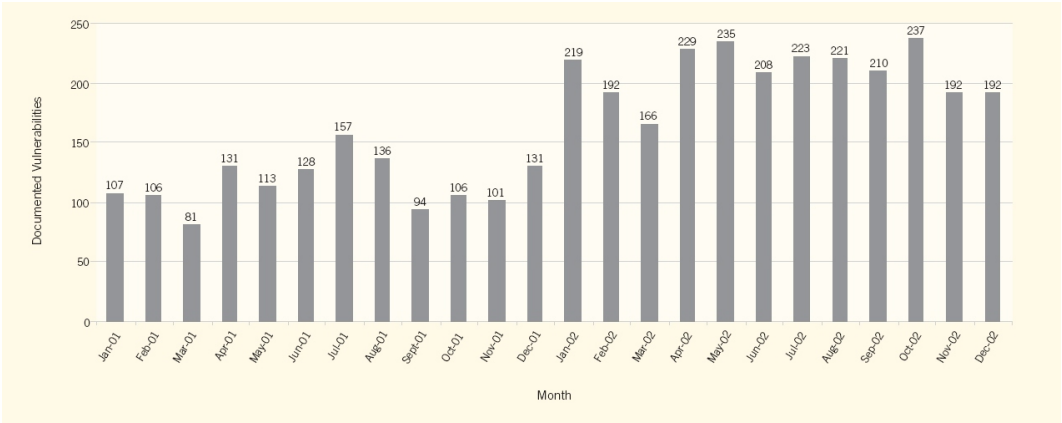

Figura 1.3: Numero di nuove vulnerabilità (Symantec, 2003)

propri sistemi, cercando inoltre di prevederne una soluzione, che comunque andrà testata e una volta che sarà risultata valida distribuita su tutti i sistemi affetti.
Nella Figura 1.4, possiamo vedere come il punto di partenza degli attacchi sia principalmente la rete Internet, seguita dalla rete interna e dagli accessi remoti. Infine è possibile concludere che internet ha permesso agli attaccanti l'accesso a numerosi sistemi informatici, portando quindi alla nascita di nuovi attacchi (malware) e alla costruzione di tool automatici per l'attacco.

1.2 Tassonomia degli attacchi e delle intrusioni

La comunità scientifica impegnata nella sicurezza informatica ha sviluppato numerose definizioni di attacchi informatici e intrusioni. Una delle definizioni di intrusione più utilizzata afferma che "una intrusione rappresenta un guasto o un difetto operativo causato da terzi con fini maligni". Intrusioni informatiche e attacchi sono spesso considerati sinonimi ma in letteratura sono state date definizioni di attacco che differenziano i due concetti. Per esempio un sistema può essere attaccato sia dall'esterno che dall'interno, ma le difese a protezione del sistema possono essere tali da rendere vano ogni tentativo e prevenire le intrusioni. Per questo motivo possiamo dire che un attacco è un tentativo di intrusione e una intrusione è il risultato di un attacco almeno in parte riuscito. Sono state fatte numerose classificazioni degli attacchi e delle intru-

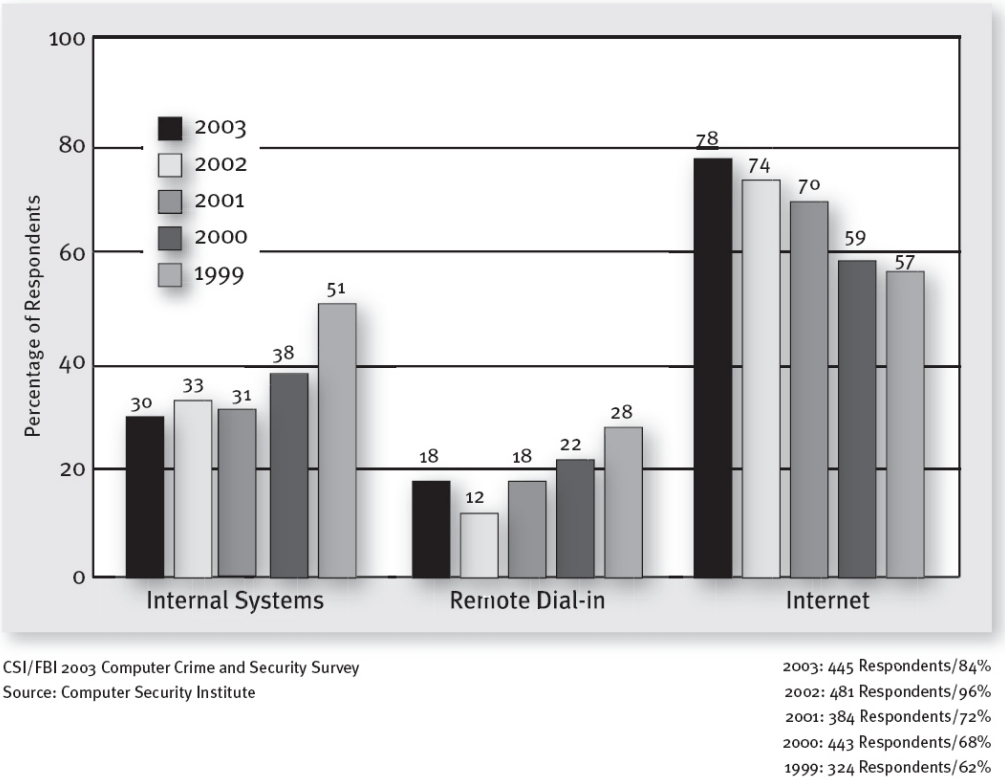

Figura 1.4: Crescita di Internet come sorgente di Attacchi (CSI/FBI 2003)

sioni; la tassonomia qui proposta è generale ed è ottenuta esaminando e combinando le categorizzazioni e le tassonomie esistenti degli attacchi ai sistemi ed alle reti, pubblicati nella letteratura che riguarda gli intrusion detection system. Altre tassonomie si basano sulle cause delle vulnerabilità o sul risultato delle vulnerabilità. La tassonomia qui proposta prevede la seguente classificazione:

- Tipo di attacco
- Numero di connessioni alla rete coinvolte nell'attacco
- Ambiente in cui avviene l'attacco
- Livello di automazione.

1.2.1 Tipi di attacco

Il più comune criterio in letteratura per classificare gli attacchi informatici e le intrusioni si basa sul tipo di attacco.

Denial of Service (DoS) : questi attacchi tentano di rendere inutilizzabile la rete o un sistema, o di impedire l'utilizzo delle risorse o dei servizi agli utenti autorizzati. Ci sono due tipi di attacchi DoS: *(i) attacchi al sistema operativo,* le cui vulnerabilità possono essere risolte applicando patch; *(ii) attacchi alla rete,* che sfruttano le limitazioni dei protocolli di rete e delle relative infrastrutture.

Probing (surveillance, scanning) : questi attacchi prevedono scansioni della rete per identificare IP validi e su questi vengono raccolte informazioni. Spesso queste informazioni forniscono a chi attacca il sistema una lista di vulnerabilità che possono essere successivamente utili per lanciare attacchi a sistemi e servizi. Questi attacchi sono probabilmente i più comuni in quanto sono i precursori ad attacchi di altro tipo.

Compromissione : questi attacchi sfruttano vulnerabilità conosciute come per esempio buffer overflow e punti di debolezza dei sistemi per ottenere un accesso ed eventualmente accrescere i propri privilegi. In base alla sorgente di attacco, interna piuttosto che esterna, la compromissione del sistema può essere classificata in queste due categorie:

R2L (Remote to Local) questi tipi di attacchi prevedono che un attaccante che è in grado di mandare pacchetti ad un sistema in rete (ma a cui non ha accesso non avendo un account) riesca ad ottenere una accesso. Esempi tipici di attacchi R2L sono: password-guessing, buffer overflow.

U2R (User to Root) questi tipo di attacchi prevedono che un attaccante con un account sul sistema sia in grado di aumentare i propri privilegi sfruttando una vulnerabilità, un bug nel sistema operativo o in una applicazione installata nel sistema. Diversamente dagli attacchi R2L, in cui l'intruso cerca di entrare nel

sistema, negli U2R l'attaccante è già un utente del sistema che intende però aumentare i propri privilegi al massimo diventando amministratore (root).

Virus/Worms/Trojan horse sono programmi che cercano di replicarsi e propagarsi tramite la rete. I Virus [16] sono programmi che si riproducono attaccandosi ad altri programmi e infettandoli. I Virus possono causare parecchi danni ai sistemi ma possono anche essere innocui e causare solo fastidiosi scherzi. I virus tipicamente necessitano di una interazione umana per replicarsi su altri computer.
Al contrario i Worm sono programmi che si replicano autonomamente e che si diffondono velocemente attraverso la rete grazie a servizi automatici di invio e ricezione di pacchetti comunemente presenti nella maggior parte dei computer. I worm possono essere divisi nelle seguenti categorie: worm tradizionali, solitamente utilizzano le comuni connessione di rete per diffondersi e non necessitano dell'intervento di nessun utente; email worm, infettano gli altri computer della rete sfruttando i contatti email dell'utente o altre applicazioni client; Windows file sharing worms, si replicano utilizzando MS Windows peer-to-peer service, che si attiva tutte le volte che nel sistema viene rilevato un nuovo device; Hybrid worms, tipicamente sfruttano diversi tipi di vulnerabilità. Alcuni recenti worm vengono anche utilizzati per attacchi di tipo DoS.
I trojan horse vengono definiti come applicazioni maligne che tentano di danneggiare la sicurezza di un sistema mascherandosi come applicazioni lecite e non dannose. Per esempio, un utente scarica dalla rete Internet una applicazione che sembra essere una gioco, ma in realtà quando il programma viene eseguito, potrebbe cancellare tutti i dati presenti sul disco o fare qualsiasi altra azione maligna.

1.2.2 Numero delle connessioni

Gli attacchi informatici possono essere classificati in base al numero di connessioni di rete coinvolte. Esempi di attacchi che coinvolgono più connessioni di rete sono DoS,

probing e worm. Gli attacchi che coinvolgono una singola connessione o un limitato numero di connessioni di rete di solito sono la causa della compromissione di un singolo sistema, come per esempio gli attacchi di tipo buffer overflow remoti. La maggior parte degli attacchi parte da un singolo sistema, ma nei casi di attacchi di Denial of Service distribuiti (DDoS) o altri attacchi organizzati, si hanno diverse sorgenti che possono partecipare all'attacco. Spesso oltre a partire da diversi sistemi, questi attacchi hanno anche diversi obiettivi. Rilevare questi attacchi distribuiti richiede uno studio e una correlazione delle informazioni ottenute dalla rete molto onerosa e non sempre efficace.

1.2.3 Ambiente in cui avviene l'attacco

Gli attacchi informatici possono essere classificati in base all'ambiente nel quale avvengono. Le intrusioni di tipo *host* sono rivolte ad un singolo sistema non connesso alla rete e possono essere rilevate analizzando le informazioni e i dati unicamente presenti sul sistema. L'identità di chi ha tentato l'attacco è tipicamente associata ad uno username, ed è quindi più semplice rilevare il colpevole.
Le intrusioni di rete *(network intrusion)* sono intrusioni che vengono effettuate tipicamente dall'esterno dell'organizzazione o dell'azienda e il rilevamento di queste avviene analizzando le informazioni contenute nel traffico di rete generato. Tuttavia queste analisi spesso non sono in grado rivelare la precisa identità di chi ha attaccato il sistema, siccome non c'è una reale ed univoca associazione tra le connessioni di rete e l'utente reale.
Le intrusioni nei sistemi peer-to-peer sono intrusioni che avvengono in sistemi che operano come peer nella rete Internet. Sebbene le applicazioni P2P di file sharing possono aumentare la produttività di una grossa rete aziendale, possono anche introdurre vulnerabilità, in quanto possono permettere agli utenti il download di codice eseguibile. Tali applicazioni possono nascondere backdoor maligne e non tracciabili sui sistemi interni e compromettere la sicurezza dell'intera azienda o organizzazione.
Le intrusioni in reti wireless possono essere generalmente rilevate analizzando il traffico che viene effettuato tra i sistemi coinvolti e il relativo punto di accesso wireless.

Tipicamente le minacce delle reti wireless possono essere separate in: *eavesdropping* (quando viene semplicemente intercettato il traffico di una comunicazione), *intrusioni* (quando si tenta di accedere al sistema o modificare i dati), *communication hijacking* (quando un finto nodo cattura tutto il traffico del canale di comunicazione, o un nodo fa da punto di accesso fasullo per attirare tutti i nodi e raccoglierne i dati confidenziali inviati), *denial of service o attacco jamming* (quando viene disturbato il canale di comunicazione con vari domini di frequenza, oggetti fisici e ostacoli portando alla caduta della comunicazione per quel o quei canali).

1.2.4 Livello di automazione

In base al livello di automazione dell'attacco possiamo avere le seguenti categorie:
Attacchi automatici: utilizzano strumenti in grado di effettuare le operazioni basilari di probing e scanning su numerosi IP in tempi brevi. Utilizzando questi tool facilmente reperibili anche un attaccante inesperto è in grado di effettuare un attacco anche sofisticato. Questi tipi di attacco sono probabilmente i più comuni al giorno d'oggi.
Attacchi semi-automatici: utilizzano script per le fasi di scanning e per compromettere il sistema o la rete, per l'installazione del codice d'attacco, e infine usano il sistema principale o master che gestisce tutti gli altri per specificare il tipo di attacco e l'indirizzo della vittima.
Attacchi manuali: prevedono una fase di scanning manuale dei sistemi vittima che tipicamente richiede molto tempo e una conoscenza approfondita e una fase in cui conoscendo le vulnerabilità si scrive il codice per l'attacco. Attacchi di questo tipo non sono molto frequenti, ma sono tipicamente più dannosi e difficili da rilevare che i semi-automatici o gli automatici, in quanto danno all'attaccante più controllo sulle risorse. Esperti e gruppi organizzati di attaccanti generalmente usano questi tipi di attacchi per intrusioni in sistemi di importanza critica.

Capitolo 2

Il concetto di IDS e Stato dell'Arte

L'approccio convenzionale per mettere in sicurezza un sistema informatico è quello di organizzare una architettura di sicurezza composta da firewall, meccanismi di autenticazione, uso di Virtual Private Network (VPN) e della crittografia, in grado di creare uno scudo protettivo che renda vano ogni tentativo di attacco. Tuttavia, quest'architettura di sicurezza quasi sempre ha inevitabili vulnerabilità e solitamente non è sufficiente ad assicurare un buon grado di sicurezza all'infrastruttura e ad evitare attacchi. Questi oltretutto sono continuamente adattati per sfruttare le nuove vulnerabilità dei sistemi spesso dovute a cattiva progettazione o carenze ed errori implementativi.
Tutto ciò ha creato il bisogno di una tecnologia di sicurezza che sia in grado di monitorare i sistemi e di individuare i tentativi di intrusione e gli attacchi. Questo sistema viene chiamato *intrusion detection* (IDS) ed è indipendente dagli altri sistemi o architetture di sicurezza.
Il National Institute of Standard and Technology (NIST) definisce un intrusion detection system come "un processo atto a monitorare gli eventi che avvengono in un sistema o il traffico di una rete, analizzando i quali è possibile rilevare segni di intrusioni, definiti come tentativi di compromissione della confidenzialità, integrità, disponibilità, o per aggirare i meccanismi di sicurezza di una sistema o di una rete."
Un IDS può essere definito anche come una combinazione di componenti software e/o hardware in grado di monitorare sistemi e di lanciare un allarme quando avviene una intrusione.

Il primo modello di intrusion detection fu sviluppato dal Dorothy Denning al SRI International nel 1983 [6], da quel momento in poi, moltissimi modelli di IDS sono stati progettati e sviluppati sia nel mondo commerciale sia nel mondo della ricerca accademica. Tuttavia questi modelli sono estremamente diversi nelle tecniche impiegate nell'acquisizione dei dati e nell'analizzare tali dati, ma molti di essi si basano su di un'architettura standard formata da componenti con i seguenti compiti:

- **Data gathering (sensori)**: si occupano della raccolta delle informazioni dai sistemi monitorati.
- **Detector (intrusion detection analysis engine)**: analizza le informazioni raccolte dai sensori per identificare attività intrusive.
- **Database (knowledge base)**: contiene le informazioni raccolte dai sensori ma in un formato preprocessato.
- **Configurator Device**: contiene le informazioni dello stato corrente dell'IDS.
- **Respose Component**: quando una intrusione viene rilevata esegue le azioni atte a segnalare l'allarme. Queste risposte possono essere automatizzate (IDS attivo) o richiedere l'intervento umano (IDS passivo).

2.1 Caratteristiche

Ci sono alcune caratteristiche che sono fondamentali per un intrusion detection come per esempio la capacità di previsione, il tempo impiegato e la resistenza ai fallimenti. La capacità di previsione in un intrusion detection misura quanto un IDS è in grado di identificare correttamente le intrusioni senza identificare una azione legittima come una intrusione (falsi positivi). A questo proposito è utile dire che esiste un grosso problema che affligge tutti gli Intrusion Detection System attuali: la presenza di falsi negativi e l'alto numero di falsi positivi che impedisce talvolta di discernere tra allarmi fasulli e allarmi veramente reali. Da parte di un attaccante è facile sfruttare il problema

dei falsi positivi per offuscare la propria traccia di attacco: è sufficiente forzare l'Intrusion Detection System a generare allarmi fasulli, per esempio scrivendo del rumore nei file di log controllati dall'host-based IDS o HIDS (quelli di syslog per esempio) o creando un flusso dati particolare verso il sensore del network-based IDS o NIDS. L'alto tasso di falsi positivi e di falsi negativi è una conseguenza implicita dell'approccio adottato dagli IDS tradizionali: troppo generico e incapace di valorizzare le caratteristiche proprie del sistema controllato. Se da un lato l'utilizzo di procedure generiche permette di utilizzare lo stesso software in contesti di rete differenti e quindi vede il favore dei responsabili commerciali attenti ad ottenere utili maggiori a scapito della qualità, dall'altro diminuisce l'efficienza dell'IDS a scapito della sicurezza dell'intero sistema informativo.

Il tempo impiegato da un IDS per rilevare una intrusione deve essere il più breve possibile in modo da permettere al sistema di reagire all'attacco prima che venga compromesso tutto il sistema, così come l'IDS dovrebbe impedire all'attaccante di modificare o alterare l'IDS stesso. Un IDS dovrebbe essere resistente agli attacchi e dovrebbe essere in grado di ripristinarsi velocemente in caso di attacco avvenuto con successo e riprendere le sue piene funzionalità nel sistema. Un IDS dovrebbe anche essere in grado di essere resistente nel caso in cui un attaccante causi un grande numero di falsi o fuorvianti allarmi che possono compromettere la disponibilità del sistema, ed evitare velocemente questi ostacoli.

2.2 Tassonomia

In passato sono stati proposti diversi metodi di classificazione degli intrusion detection, ma non c'è una tassonomia universalmente accettata.
Per classificare gli intrusion detection system ho utilizzato cinque criteri: **la sorgente di informazioni**, che distingue gli IDS basandosi su dati che il sistema utilizza per rilevare le intrusioni, per esempio i file di log piuttosto che il traffico di rete; la **strategia di analisi** che descrive le caratteristiche del detector; **gli aspetti legati al tempo**, usati per differenziare gli IDS in on-line o real-time da quelli off-line; **l'architettura**

che distingue gli IDS in centralizzati e distribuiti; **la risposta** che descrive le azioni intraprese a fronte di una intrusione dall'IDS.

2.2.1 Sorgente di informazioni

I primi intrusion detection system erano per la stragrande maggioranza host-based, cioè legati al singolo sistema, siccome i sistemi più diffusi erano i mainframe e tutti gli utenti del sistema erano locali. Le informazioni raccolte dai mainframe venivano analizzate o localmente o in sistema separato e venivano identificati e rilevati gli eventi sospetti. Tuttavia con la diffusione delle reti c'è stato un aumento di interesse nel campo degli IDS per permettere la comunicazione di informazione tra vari host-based IDS a vari livelli, ma soprattutto per l'uso del traffico di rete generato come fonte primaria di informazione per rilevare tentativi di intrusione.
Negli anni novanta, la comunità scientifica che si occupava di intrusion detection discusse e dichiarò la superiorità dell'approccio network-based rispetto a quello host-based. Tuttavia, oggi molti modelli cercano di integrare entrambi le varianti host e network in un unico modello definito ibrido.
Come detto gli approcci network e host tipicamente analizzano il traffico di rete o le attività del sistema operativo, ma non sono in grado di rilevare un uso non autorizzato di una specifica applicazione, infatti questo tipo di allarme viene dato dal modello di IDS chiamato *application-based*, che monitora le interazioni tra uno specifico utente e una specifica applicazione.
Gli *host based intrusion detection system* (HIDS) analizzano le attività degli utenti e il loro comportamento su di uno specifico sistema con il grande vantaggio di poter avere informazioni tipicamente molto dettagliate e precise. Tuttavia, in base alle azioni svolte, gli host based IDS possono intaccare e significativamente far diminuire le performance del sistema che stanno monitorando. L'analisi delle informazioni raccolte effettuata off-line o con un certo ritardo è pericolosa e può rendere vana la protezione offerta dall'IDS, si pensi infatti ad un attacco riuscito in cui l'attaccante sia in grado di modificare i dati, che verranno utilizzati dall'IDS per la detection, in modo tale da non rilevare l'attacco. Per evitare questi svantaggi, gli HIDS devono essere in grado

di analizzare le informazioni raccolte in tempi sufficientemente veloci per lanciare un allarme prima che un attaccante abbia l'opportunità di leggere e/o modificare le informazioni utilizzate dall'IDS stesso per generare gli allarmi. Ci sono diversi tipi di informazioni che sono tipicamente utilizzate dagli IDS: i comandi e gli account di sistema, il syslog, e altre informazioni di audit riguardanti la sicurezza. Analizzando i comandi di sistema che l'utente invoca nelle sue sessioni, è possibile costruire il profilo dell'utente, che descrive le caratteristiche e il comportamento abituale dell'utente stesso. Gli account di sistema presenti sia nei sistemi operativi MS Windows che Unix e le informazioni generate dal sistema che gestisce questi account possono essere utilizzati come sorgenti di informazioni. Su Unix è disponibile un servizio di accounting che consente la raccolta di dati sull'utilizzo del sistema. L'accounting registra su log ogni comando Unix attivato, l'utente che lo ha attivato, la quantità di CPU e memoria utilizzati, e le loro aggregazioni per utente/periodo. Oltre che per ragioni di accounting tali log possono essere utilmente sfruttati per analizzare eventuali attività anomale presenti sul sistema (ad esempio un numero elevato di processi di login indica una serie di tentativi di accesso falliti). L'uso dei file di log come sorgente di informazione per gli IDS ha diversi vantaggi. Tutti i sistemi Unix hanno lo stesso formato o è comunque possibile modificarlo a proprio piacimento e il tempo per registrare tali informazioni è generalmente brevissimo in quanto le informazioni sono compresse. Tuttavia ci sono alcuni svantaggi che limitano il loro utilizzo nelle applicazioni legate alla sicurezza. Per effettuare una analisi in real-time delle informazioni degli account, tutte le informazioni precedenti devono essere confrontate con il profilo corrente, e questo può essere computazionalmente costoso e potrebbe modificare il carico del sistema e diminuire la potenziale capacità di analizzare le informazioni e quindi rilevare altri attacchi o anomalie. Inoltre i log richiedono molto spazio su disco e per questo devono essere periodicamente rimossi. Per questi motivi i log del system accounting non sono comunemente utilizzati come fonte di informazione. I log di sistema contengono informazioni che non sarebbero disponibili catturando il traffico di rete, così come tra le informazioni di accounting degli utenti, o tra i log delle applicazioni o dei servizi installati e attivi sul sistema.

La cattura e l'archiviazione delle informazioni e il relativo salvataggio in formato leg-

gibile è normalmente effettuato da un demone chiamato syslog, in passato però le informazioni raccolte da tale demone sono state ritenute poco sicure in quanto syslog era affetto da vari buffer overflow.
Le informazioni riguardanti le sicurezza che il sistema registra contengono informazioni rilevanti sulle attività svolte dal sistema. Siccome queste attività vengono generalmente salvate su file e ordinate cronologicamente, dalla loro analisi sono facilmente individuabili pattern che identificano una intrusione.

Con la rapida espansione di Internet e della connettività, c'è stato un aumento spropositato del numero degli attacchi indirizzati alla rete stessa, che non possono essere rilevati esaminando solamente le informazioni e i log raccolti nel sistema. Per queste ragioni si è cercato di sviluppare apposite applicazioni che catturano il traffico di rete e in real time analizzano i pacchetti cercando in qualche maniera di rilevare attacchi. Per esempio analizzando il solo payload dei pacchetti, parecchi attacchi possono essere rilevati. Ci sono parecchi vantaggi nell'usare i *network based intrusion detection system* (NDIS). I NIDS possono essere installati in qualunque architettura siccome non causano nessun tipo di effetto ai sistemi e alle infrastrutture esistenti; sono inoltre più resistenti rispetto agli HIDS in quanto non sono fisicamente installati sugli stessi computer che posso essere le vittime degli attacchi. La maggior parte dei NIDS non dipendono dal sistema operativo che è utilizzato per estrarre informazioni a livello rete e possono essere installati in posizioni strategiche nella rete. Tuttavia il loro più grande svantaggio è la poca scalabilità, così come la possibilità di non riuscire a catturare tutti i pacchetti su reti veloci e a pieno carico, e l'impossibilità di effettuare l'analisi dei pacchetti quando questi sono cifrati.

2.2.2 Strategie di analisi: misuse detection e anomaly detection

Ci sono due approcci per analizzare gli eventi ricostruiti dalle informazioni raccolte: analizzare le anomalie *(anomaly detection)* o analizzare gli abusi *(misuse detection).* L'approccio *Misuse detection* si basa su una precisa conoscenza degli attacchi e delle vulnerabilità dei sistemi costantemente aggiornate e corrette dagli esperti.

Questo approccio si basa sulla ricerca di sequenze di eventi conosciuti che indicano l'esecuzione di un attacco o del tentativo di sfruttare vulnerabilità conosciute. Tale approccio può essere molto accurato nel rilevare gli attacchi conosciuti, ma nulla può con attacchi nuovi o semplici minacce. L'approccio anomaly detection è invece basato sull'analisi del comportamento normale degli utenti, sistemi, applicazioni, connessioni di rete e sulla creazione di profili, questa fase viene detta fase di learning. Con tale approccio si tenta di caratterizzare il comportamento normale e legittimo del sistema monitorato; questo verrà poi confrontato in una fase di detection con il comportamento rilevato in real-time e ogni deviazione che supera una certa soglia verrà rilevata come una potenziale intrusione o attacco. Il maggior vantaggio dato da questo tipo di approccio è dato dalla possibilità di rilevare attacchi sconosciuti, ma tuttavia le maggiori limitazioni sono date dal fatto che esistono molti falsi positivi. Infatti una deviazione dal comportamento normale non significa necessariamente che è avvenuto un attacco. Gli approcci misuse possono essere classificati nelle seguenti quattro categorie:

- metodi basati sulle signatures
- metodi basati sull'analisi delle transizioni di stato
- tecniche rule-based
- tecniche basate sul data mining

Gli intrusion detection basati sulle signature operano in maniera simile agli scanner di virus; gli eventi monitorati vengono identificati da delle stringhe univoche che vengono confrontate con quelle registrate nei database degli attacchi. Questi tipi di IDS non sono in grado di rilevare nuovi tipi di attacchi finché il database non viene aggiornato con l'inserimento delle signature dei nuovi attacchi.

I sistemi basati su regole sono composti da una serie di regole di implicazione del tipo "if-then" che caratterizzano gli attacchi informatici. Nei primi sistemi di intrusion detection, il linguaggio basato su regole era un normale metodo attraverso il quale venivano descritte le conoscenze degli esperti circa attacchi e vulnerabilità. Nei sistemi basati su regole vengono normalmente monitorati gli eventi che sono poi convertiti in

azioni e regole che verranno poi utilizzate da un motore inferenziale per decidere le azioni da intraprendere.

I sistemi di intrusion detection basati sull'analisi delle transizioni di stato modellano gli attacchi come azioni che causano transizioni nello spazio degli stati di sicurezza di un sistema. Tutte le volte che l'automa raggiunge uno stato che è stato marcato come una minaccia alla sicurezza del sistema, l'IDS segnala la possibile intrusione.

I sistemi di intrusion detection basati sul data mining prevedono l'estrazione di informazioni utili, eseguita in modo automatico o semiautomatico, da grandi quantità di dati raccolti e l'identificazione dei comportamenti normali o intrusivi da queste informazioni. Questo tipo di intrusion detection sono in grado di rilevare attacchi se la fase di apprendimento automatico o di learning è stata eseguita in maniera appropriata. La ricerca scientifica nel campo dei misuse intrusion detection system si è focalizzata principalmente nella classificazione delle intrusioni di rete usando vari algoritmi standard di data minig. Diversamente dai signature-based intrusion detection systems, i modelli basati sul data mining sono creati automaticamente e possono essere molto più complicati e precisi che le signatures inserite manualmente. Il vantaggio dei modelli basati sul data minig rispetto a quelli basati sulle signatures è il loro altissimo livello di accuratezza nel rilevare gli attacchi conosciuti e le loro variazioni.

L'aumento del numero di attacchi informatici e della loro complessità ha accresciuto l'interesse negli algoritmi basati su anomalie siccome questi sono in grado di riconoscere nuovi tipi di attacchi e tentativi di intrusione. In letteratura sono stati proposti vari modelli di *anomaly detection* che si differenziano per le informazioni utilizzate per l'analisi e per le tecniche utilizzate per rilevare le deviazioni dai comportamenti normali.

Le tecniche di rilevamento di intrusione basate su anomalie possono essere classificate nei seguenti quattro gruppi:

- metodi statistici

- metodi basati su regole
- metodi basati sui profili
- metodi basato sui modelli

Gli algoritmi utilizzati nei modelli anomaly based sono numerosi, e possono essere inseriti in più di una categoria; con questa classificazione si cerca di trovare la migliore categoria per ciascuno degli algoritmi considerati.

I metodi statistici monitorano il comportamento degli utenti e del sistema misurando certe variabili nel tempo. Il modello più semplice calcola le medie di queste variabili e rileva se le soglie sono state superate basandosi sulle deviazioni standard delle variabili. Modelli statistici più complessi confrontano i profili degli utenti e delle attività dei processi. Questi modelli statistici sono usati negli host based IDS, così come nei network based IDS.
Molti degli approcci statistici hanno limitazioni nel momento in cui si trovano a dover rilevare delle deviazioni in spazi multidimensionali siccome diventa sempre più difficile e poco preciso dare una stima di alcuni punti in una distribuzione multidimensionale.

I modelli basati su regole nell'approccio anomaly based caratterizzano il comportamento normale degli utenti della rete e del sistema con un insieme di regole. Queste regole sono poi passate ad un sistema che analizzando i dati raccolti è in grado di rilevare eventuali violazioni delle regole. Se le regole violate indicano un comportamento anomalo si allerta l'analista o si genera un allarme.

I metodi basati sui profili creano i profili dei comportamenti normali del sistema o del traffico di rete, e le deviazioni da questi sono considerati come intrusioni. I metodi per creare i profili differiscono molto tra loro e vanno dal data mining alle tecniche euristiche.
Sono state sperimentate anche tecniche basate sulle system call: vengono creati dei pattern che rappresentano il comportamento normale di un servizio, cioè la lista di system call eseguite e questi pattern verranno poi utilizzati in tempo reale per rilevare

anomalie e deviazioni rispetto alle sequenze imparate.
Per creare profili degli utenti sono stati calcolati perfino i tempi e le frequenze con cui venivano eseguiti i comandi dalla shell e utilizzati per rilevare comportamenti anomali. Come vedremo nei prossimi capitoli, oltre alle system call anche le informazioni contenute nello stack possono essere utilizzate per caratterizzare il comportamento e creare profili.

2.2.3 Aspetti legati al tempo

Quando si considerano i tempi negli IDS è necessario distinguere due tipi di modelli: i modelli *real-time o on-line* e i modelli *off-line*.
Gli IDS real-time sono quegli IDS che continuamente analizzano le informazioni fornitegli e che riescono a rilevare velocemente comportamenti insoliti o sospetti. Gli IDS real-time devono inoltre essere in grado di lanciare un allarme non appena viene rilevato un attacco, così che l'azione associata a quel tipo di attacco possa essere compiuta.
Gli IDS off-line eseguono l'analisi dei dati raccolti in un momento successivo alla raccolta. Questo metodo di analisi dei dati è comune tra gli analisti che effettuano controlli periodici del sistema piuttosto che della rete. Molti dei primi IDS usavano questo metodo di analisi in quanto utilizzavano come sorgente di informazioni i log del sistema che venivano scritti su disco come file. L'analisi off-line viene effettuata con strumenti statici che analizzano uno snapshot del sistema, cercando vulnerabilità ed errori di configurazione.

2.2.4 Architettura

Le principali architetture degli IDS sono due, *centralizzato* o *distribuito*.
Molti IDS utilizzano una architettura centralizzata e rilevano le intrusioni che avvengono a livello del singolo host monitorato. Tuttavia sono numerosi gli attacchi che vengono organizzati e coordinati per coinvolgere numerosi sistemi sia come attaccanti che come vittime. Le informazioni utilizzate comunemente dagli HIDS non sono in

grado di permettere la previsione e il rilevamento di questi tipi di attacco. Per rilevare questi attacchi è necessaria la cooperazione di molti IDS e l'analisi del traffico di rete. Nonostante i vari problemi che affliggono i sistemi distribuiti, molte aziende hanno sviluppato IDS in grado di coordinarsi per poter rilevare ancora più tipi di attacchi ed essere il più precisi possibile.

2.2.5 Risposta agli attacchi

Le azioni che possono essere intraprese da un IDS quando viene identificato un attacco possono essere sia *attive* che *passive*.
Lo scenario più comune prevede un IDS con risposta passiva, cioè in caso di intrusione rilevata viene semplicemente informato il responsabile di turno o semplicemente viene scritto su file qualche tipo di messaggio, ma non viene presa nessun tipo di contromisura.
Diversamente altri tipi di IDS prevedono una risposta attiva in caso di evento critico, come per esempio applicando una patch alla vulnerabilità, piuttosto che effettuando il log-off dell'utente non autorizzato, riconfigurando i firewall o i router e disconnettendo una porta. Data la velocità e la frequenza con cui gli attacchi avvengono un IDS ideale dovrebbe automaticamente rispondere agli attacchi senza l'intervento di nessun operatore. Tuttavia questo è irrealizzabile dato l'alto numero di falsi positivi.
Nonostante ciò, molti IDS prevedono la possibilità di utilizzare un meccanismo attivo di risposta, che può essere utilizzato a discrezione dell'amministratore di sistema. Uno dei più semplici ma spesso molto utile meccanismo attivo di risposta è quello di raccogliere informazioni aggiuntive su ogni sospetto di attacco ed effettuare un controllo per assicurarsi che non ci siamo stati dei danni. Le informazioni raccolte potrebbero aiutare l'IDS a rilevare un futuro attacco ed eventualmente a scoprire l'identità dell'attaccante. Nei prossimi paragrafi focalizzerò l'analisi sugli host base intrusion detection system (HIDS).

2.3 Stato dell'arte degli HIDS

Molti ricercatori si sono concentrati nello studio e nell'implementazione di modelli di anomaly detection cercando di imparare il comportamento dei programmi. Molti dei metodi recentemente proposti si basano sulla modellizzazione delle tracce delle syscall eseguite.

Forrest et al. [10] ha introdotto un nuovo approccio per il rilevamento di intrusioni che identifica e segnala sequenze anomale di system call che vengono eseguite da una applicazione. Questo modello si basa sull'intuizione che il comportamento di un programma p può essere caratterizzato dalla sequenza di system call che questo esegue durante la sua esecuzione in una ambiente protetto, cioè dove non può subire attacchi e intrusioni. Nel modello le sequenze di system call, chiamate N-gram, vengono salvate in un database e rappresentano un linguaggio l che caratterizza il normale comportamento di p. Per rilevare le intrusioni, durante l'esecuzione del processo vengono collezionate le sequenze di system call di una data lunghezza e vengono confrontate con quelle salvate nel database. Vengono calcolate le distanze di Hamming[1] tra le stringhe catturate e quelle del linguaggio l e quando si supera una certa soglia, viene segnalata una anomalia con il lancio di un allarme da parte dell'HIDS. Il modello N-gram è molto semplice ed efficiente ma presenta un alto numero di falsi positivi, principalmente perché non viene registrata l'informazione della posizione dalla quale la syscall viene invocata. Un modo semplice e naturale per registrare le sequenze, in questo caso di system call, prevede l'utilizzo di un automa a stati finiti (FSA). Tuttavia ricerche precedenti ritengono che l'uso di automi a stati finiti per questi scopi sia computazionalmente troppo costoso, che questo processo non possa essere completamente automatizzato, e che lo spazio utilizzato dall'automa sia troppo. Recentemente non ci sono stati molti miglioramenti dei metodi basati sulle system call, in parte perché le syscall stesse forniscono solamente una quantità di informazioni limitata e l'esecuzione delle syscall è solamente uno degli aspetti del comportamento di una applicazione. Wagner et al. [20] hanno proposto un modello che genera staticamente un automa a

[1] La distanza di Hamming misura il numero di sostituzioni necessarie per convertire una stringa nell'altra, o il numero di errori che hanno trasformato una stringa nell'altra.

stati finiti non deterministico (NDFA) dal control-flow graph dell'applicazione. Questo automa non è deterministico perché in generale non può essere staticamente previsto quale ramo verrà scelto. L'automa viene poi usato per monitorare l'esecuzione dell'applicazione in real-time. Le operazioni del NDFA vengono simulate non-deterministicamente basandosi sulla lista delle syscall osservate e se tutti i possibili percorsi non-deterministici sono bloccati, viene rilevata una anomalia. Con questo metodo non è possibile avere falsi allarmi siccome tutti i possibili percorsi di esecuzione vengono considerati dall'automa. Il problema principale di questi tipi di modelli è dato dall'eccessiva lentezza della fase di monitoraggio. L'overhead di questo modello è risultato essere di circa 40 minuti per transazione per circa la metà delle applicazioni monitorate dagli autori durante i loro esperimenti. Questo overhead è dovuto alla complessità dell'automa a pila e al non determinismo della simulazione.

Giffin et al. [8] hanno proposto alcune migliorie al modello di Wagner [20] in particolare utilizzando l'analisi statica sui binari eseguibili, essendo così indipendenti dai linguaggi di programmazione utilizzati. Inoltre hanno migliorato la precisione e l'efficienza del modello sviluppando numerose tecniche di ottimizzazione e offuscamento. In particolare con l'inserimento nell'eseguibile di chiamate nulle "null call" sono in grado di abbassare il livello di non determinismo e quindi aumentare la precisione. Queste tecniche richiedono la riscrittura degli eseguibili e la modifica delle chiamate a funzione. Nell'articolo in cui presentano il modello vengono riportati bassi valori di overhead riscontrati nei loro esperimenti.

Sekar et al. [15] hanno proposto un modello in grado di generare un automa a stati finiti FSA compatto, monitorando l'esecuzione dell'applicazione a runtime senza riscontrare problemi di non determinismo. Questo modello, al contrario dei precedenti, è in grado di catturare le relazioni temporali tra le system call e quindi effettuare un rilevamento più accurato. L'FSA proposto è in grado di catturare le strutture basilari della programmazione come per esempio salti,unioni,cicli ecc. Questo modello è in grado di astrarre e prevedere il comportamento da quelli passati.

Feng et al. [7] estendono il lavoro di Sekar [15] imparando le sequenze di system call e registrando il loro contesto. Il loro modello chiamato Virtual Path prevede la collezione di tutte le coppie di system call eseguite, di alcune informazioni prese dallo stack e

dei cambiamenti avvenuti nello stack nel passaggio da una all'altra. Queste informazioni vengono raccolte tracciando un processo durante le sue esecuzioni e salvate in un database. Un'evoluzione del modello Virtual Path è stato presentato da Giffin [9], tale metodo basandosi su informazioni più precise riesce a rilevare attacchi diversamente non rilevabili. Tutti questi modelli che sono stati presentati vengono proposti come syscall-based. Tuttavia la cosa interessante è che implicitamente o esplicitamente per costruire gli stati degli automi viene usata l'informazione ottenuta dal program counter.

2.3.1 Problemi

A parte più o meno gravi problemi di performance, questi modelli presentano difetti che possono essere sfruttati da un attaccante per aggirare l'IDS e nullificare la protezione voluta. Wagner et al. [20] hanno dimostrato che alcuni modelli non sono in grado di rilevare alcuni tipi di attacchi, in particolare alcune forme di Impossible Path Execution (IPE), ed inoltre negli articoli [20, 12] viene dimostrato che tutti i precedenti modelli presentati non sono in grado di rilevare alcuni tipi di attacchi chiamati mimicry.

IPE

Un "impossible path" può essere definito come una sequenza di istruzioni che non può mai essere eseguita in circostanze normali data la particolare struttura dell'applicazione. Un esempio tipico di questa situazione lo si può trovare nel costrutto "if() then ... else ...". Se la CPU esegue alcune istruzioni nel ramo true, non c'è nessun modo di saltare nel ramo false indipendentemente da quante iterazioni si facciano. Questo passaggio impossibile è dovuto alla semantica del costrutto if/then/else. Se individuato da un attaccante, un impossible path può essere sfruttato per eseguire il codice dell'applicazione in un modo che non dovrebbe essere possibile; per esempio i controlli di sicurezza potrebbero essere saltati. Questi tipi di attacchi che forzano l'esecuzione di una sequenza di istruzioni altrimenti impossibile vengono chiamati Impossible Path

Execution (IPE).
Alcuni modelli di HIDS sono in grado di rilevare alcuni ma non tutti i tipi di attacchi IPE, come mostrato in [23, 7]. Nella Figura 2.1 troviamo un esempio di codice presentato in [7] e leggermente modificato per mostrare ancora meglio come questi attacchi non vengano rilevati dalla maggior parte degli HIDS, tra i quali ricordiamo [15, 20, 10]. Si supponga che la funzione `is_regular(uid)` (linea 20) invochi la syscall `open` due volte per aprire rispettivamente i file `/etc/passwd` e `/etc/group` per controllare l'`uid` dell'utente e decidere se è valido o meno (parte non implementata). Se l'utente rappresentato dall'`uid` non ha particolari privilegi verrà seguito il ramo *true*, altrimenti l'esecuzione prosegue seguendo il ramo *false*. Un impossible path si potrebbe verificare seguendo il ramo *true* e "saltando" nel ramo *false*. In particolare un utente regolare entrando nel ramo *true* del costrutto `if` (linee 21-27) sfruttando uno stack-based buffer overflow nella funzione `read_next_cmd` alla linea 8, è in grado di cambiare il flusso di esecuzione del programma *p* ed entrare nel ramo *false* dove può ottenere tutti i privilegi[2]. Certamente è possibile obiettare che gli attacchi di tipo IPE sono possibili solamente in condizioni particolari, struttura del programma, vulnerabilità nella "giusta posizione", stesse sequenze di system call per tutta l'esecuzione ecc., ma come detto da Feng et al. [7] non vanno tralasciati siccome è abbastanza facile per un attaccante introdurre volontariamente le giuste condizioni nel codice sorgente per permettere l'esecuzione di un impossible path.

Mimicry

L'attacco di tipo mimicry è stato descritto per la prima volta da Wagner et al.[21, 20] come un attacco che poteva essere eseguito sui modelli di HIDS basati sulle system call. La sua forma più semplice chiamata *traditional mimicry* prevede l'esecuzione "fasulla" della sequenza delle syscall salvata dall'HIDS. In altre parole, vengono eseguite tutte le syscall che sono contenute nella traccia, altrimenti l'HIDS genererebbe una anomalia, ma tutte le syscall non necessarie all'attaccante per lanciare l'attacco

[2]Per semplicità assumiamo che la funzione di libreria `system` invochi solamente la system call `execve`.

vengono "nullificate". In pratica si simula il comportamento dell'applicazione, ma in realtà si esegue il codice dell'attacco.

Traditional Mimicry Attack

Il Traditional Mimicry Attack può essere eseguito se sono soddisfatte le seguenti due assunzioni:

1. l'attaccante conosce le sequenze delle system call eseguite dal processo e salvate nel database

2. l'attaccante ha il pieno controllo del flusso di esecuzione non appena è in grado di eseguire il codice iniettato [13] o già presente [22].

A parte queste assunzioni, l'attaccante può scegliere la traccia che è più conforme al codice del suo attacco, e costruire un injection vector che gli permetta in qualche maniera di eseguire la traccia di syscall scelta. Questa traccia conterrà syscall nullificate che non produrranno nessun effetto e syscall che eseguiranno il codice dell'attacco voluto che permetteranno di ottenere il controllo del sistema. Come mostrato nella Figura 2.2 solamente le syscall necessarie all'attacco saranno eseguite con successo mentre le altre verranno nullificate semplicemente facendole fallire per esempio passandogli i parametri scorretti.
Non è complicato dimostrare la veridicità della seguente affermazione.

Affermazione 1 (Traditional Mimicry Attack) *Un attaccante A può eseguire un attacco di tipo traditional mimicry su di un processo p solo se conosce le tracce delle system call eseguite da p.*

Automating Mimicry Attack

Uno dei punti fondamentali dell'esecuzione e della riuscita del traditional mimicry è la possibilità da parte dell'attaccante di controllare l'esecuzione dell'applicazione. Infatti è abbastanza semplice per un attaccante eseguire il codice desiderato sul sistema

vittima[3]. Recentemente Kruegel *et al.* [12] hanno presentato una variante dell'attacco traditional mimicry e hanno sviluppato uno strumento *proof of concept* che rende capace un attaccante di avere il pieno controllo del flusso di esecuzione dell'applicazione modificando dei particolari code pointer, evitando anche i controlli sulle syscall. Le vulnerabilità vengono sfruttate sovrascrivendo particolari code pointer, come il return address, le entry della Global Offset Table (GOT) [4], puntatori a funzione, longjmp buffers ecc.

[3]Assumendo che il sistema operativo non abbia particolari meccanismi di protezione come per esempio l'Address Space Layout Randomization (ASLR) [19, 5, 17] o la non-executable data area [18, 11, 19]

[4]Un eseguibile dynamically-linked ELF utilizza la Procedure Linkage Table (PLT) e la Global Offset Table (GOT) per chiamare le funzioni di libreria. L'applicazione esegue una chiamata diretta alla entry della funzione di libreria PLT che chiama indirettamente la funzione di libreria rilocata grazie al valore salvato nella entry della GOT risolta dal run-time dynamic linker (rtdl). Lo rtdl risolve i simboli così che le prossime chiamate siano eseguite senza richiamare lo rtdl, ma semplicemente usando l'indirizzo salvato nella corrispondente entry nella GOT. La sovrascrittura di queste entry rendono possibile e semplice il code hijacking.

```
 1: u_char *read_next_cmd(void) {
 2:
 3:    u_char input_buf[64];
 4:    u_char *p;
 5:
 6:    umask(2);
 7:    ...
 8:    strcpy(&input_buf[0], getenv("USERCMD"));
 9:    /* memory leak? :-) */
10:    p = (char *)strdup(input_buf);
11:    return p;
12: }
13:
14: void login_user(int uid) {
15:
16:    char *cmd;
17:
18:    /* why do you call it "poor programming style"?! :-) */
19:
20:    if (is_regular(uid)) {
21:
22:       /* unprivileged mode */
23:       cmd = read_next_cmd();
24:       setuid(uid);
25:       /* yes, system is safe ;-) */
26:       system(cmd);
27:
28:    }
29:    else {
30:
31:       /* superuser! */
32:       cmd = read_next_cmd();
34:       setuid(0);
35:       system(cmd);
36:
37:    }
38:    return;
39: }
```

Figura 2.1: Codice sorgente che può essere sfruttato per un attacco IPE.

sequenza normale: $S_1\ S_2 \mid S_3\ S_4\ S_5\ S_6$

La *sequenza normale* è la sequenza eseguita dal processo e imparata dall'IDS, $\mid$ rappresenta il punto dove è presente la vulnerabilità e S_i rappresenta una generica syscall *i*.

sequenza dell'attacco: $S_3^s\ S_4^s\ S_5\ S_6^s$

La *sequenza dell'attacco* è la sequenza costruita dell'attaccante e comprende le syscall "nullified" (S_i^s), così come le system call che l'attaccante vuole eseguire (S_5).

Figura 2.2: Traditional Mimicry Attack

Capitolo 3

Modello VTPath

Nell'articolo Anomaly Detection Using Call Stack Information presentato da H. Feng, O. Kolesnikov, P. Fogla, W. Lee e W. Gong al Symposium on Security and Privacy di Oakland, California nel 2003 viene proposto un metodo di rilevamento delle intrusioni basato sulle anomalie utilizzando le informazioni dello stack. L'idea alla base del modello è quella di estrarre informazioni dallo stack e nella fase di learning generare un percorso astratto di esecuzione tra ogni coppia di syscall eseguite dal programma, e salvarlo in una forma compatta in un database. In una successiva fase di detection si deciderà in real-time se ogni percorso è valido cercando deviazioni da quelli che sono stati acquisiti in fase di learning.

3.1 Il modello

Il modello presentato possiede parecchie proprietà non presenti nel metodo presentato da Sekar et al. [15]: utilizza informazioni ottenute dallo stack e il valore del program counter, è in grado di percorrere lo stack delle funzioni eseguite e risolvere alcuni problemi implementativi come per esempio la gestione dei segnali.
Il metodo di Sekar et al. [15] e il VTPath prevedono il salvataggio del valore del program counter all'esecuzione di ogni system call, siccome non è necessario né possibile, se si vuole mantenere una efficienza tollerabile, registrare il program counter di ciascuna istruzione eseguita. Questa è risultata essere una scelta valida siccome le system

call sono quelle funzioni che permettono al programma di interagire con il kernel. Il valore del program counter così come gli altri registri dello stack posso essere estratti con un basso overhead all'esecuzione di ogni system call.

3.2 Virtual Stack List e Virtual Path

Il modello prevede che ogni volta che viene eseguita una system call, venga estratto il nome o il numero della system call, il valore del program counter al momento di esecuzione della system call e tutti gli indirizzi di ritorno delle funzioni non ancora ritornate fino ad arrivare alla cima dello stack. Queste informazioni verranno poi inserite in una struttura dati chiamata Virtual Stack List.
Un Virtual Stack List viene rappresentato come:

$$A = \{a_0, a_1, ..., a_{n-1}, a_n\}$$

dove n è il numero dei record di attivazione presenti sullo stack e a_{n-1} è l'indirizzo di ritorno della funzione che è stata chiamata per ultima e a_n è il program counter della funzione corrente.
Per esempio si assuma che la funzione somma() sia chiamata nella funzione main(). Il Virtual Stack List contiene tre elementi: a_0 e a_1 sono gli indirizzi di ritorno di $main()$ e $somma()$, rispettivamente; a_2 è il program counter corrente.
Il Virtual Stack List rappresenta la storia di tutte le funzioni non ancora ritornate.

```
int somma()
{
...
}
int main()
{
somma;
}
```

Per rappresentare la transizione tra due system call il modello utilizza il concetto di Virtual Path. Si assuma che $A = \{a_0, a_1, ..., a_{n-1}, a_n\}$ e $B = \{b_0, b_1, ..., b_{n-1}, b_n\}$ siano i Virtual Stack List della syscall corrente e della precedente, rispettivamente. Si confrontano gli indirizzi contenuti delle liste A e B dal primo valore fino a quando si trova il primo record di attivazione l tale per cui $a_l \neq b_l$.
Come mostrato nella Figura 3.1, il Virtual Path tra le due system call è definito come:

$$P = b_m \rightarrow Exit; ...; b_{l+1} \rightarrow Exit; b_l \rightarrow a_l; Entry \rightarrow a_{l+1}; ...; Entry \rightarrow a_n$$

dove *Entry* e *Exit* sono due speciali valori del program counter che denotano i punti di ingresso e uscita di ciascuna funzione.

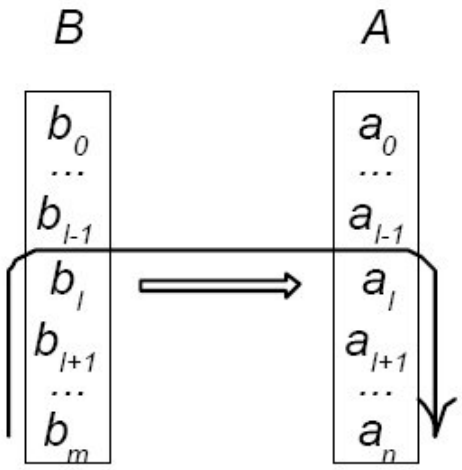

Figura 3.1: Il Virtual Path dall'ultima syscall B alla syscall corrente A.

La definizione data del Virtual Path astrae il percorso di esecuzione tra due system call: l'applicazione ritorna sequenzialmente da alcune funzioni (da b_{m-1} a b_l) e gradualmente entra in altre funzioni (da a_l a a_{n-1}).
I Virtual Stack List ottenuti da funzioni ricorsive possono risultare diversi in base ai parametri che vengono passati, il che porterebbe ad un elevato numero di Virtual Stack List oltre che ad un alto numero di Virtual Path. Questo complicherebbe la fase di learning e porterebbe in fase di detection ad un alto numero di falsi positivi.

Siccome in caso di ricorsione è noto che nei Virtual Stack List sono presenti più volte gli stessi indirizzi, è possibile effettuare una semplificazione. Quando nel Virtual Stack List è presente una coppia degli stessi indirizzi, tutti gli indirizzi di ritorno compresi tra i due incluso una delle due estremità vengono rimossi dalla lista. In questo modo le funzioni ricorsive invocate con parametri diversi vengono identificate tutte dagli stessi Virtual Stack List.

3.3 Fase di Learning e di Detection

Durante la fase learning tutti gli indirizzi di ritorno che hanno permesso la costruzione dei Virtual Stack List vengono salvati con il numero delle relative syscall e il valore del program counter in una tabella di un database.
Per salvare i Virtual Path viene utilizzata un'altra tabella dello stesso database.
I Virtual Stack List e i Virtual Path se non sono già presenti vengono gradualmente inseriti nel database durante le normali esecuzioni del programma. Per ciascuna esecuzione c'è una system call nulla con un Virtual Stack vuoto prima dell'esecuzione della prima system call, e un'altra dopo l'ultima system call realmente eseguita.
Il Virtual Path tra la system call nulla e la prima system call reale, il cui Virtual Stack è $A = \{a_0, a_1, ..., a_n\}$, è dato da:

$$P = Entry \rightarrow a_0; ...; Entry \rightarrow a_n$$

Il Virtual Path tra l'ultima reale system call,il cui Virtual Stack è $B = \{b_0, b_1, ..., b_m\}$ e la syscall nulla, è dato da:

$$P = b_m \rightarrow Exit; ...; b_0 \rightarrow Exit;$$

Una volta terminata la fase di learning i dati raccolti vengono utilizzati per monitorare il processo durante la sua esecuzione.
Non appena viene eseguita una syscall vengono costruiti il Virtual Stack List e il Virtual Path e confrontati con quelli salvati nel database durante la fase di learning.

Ci possono essere vari tipi di anomalia:

- Se non è possibile accedere ai record di attivazione delle funzioni nello stack è ipotizzabile che questo sia stato corrotto, e viene rilevata una Stack Anomaly. Questo tipo di anomalia tipicamente avviene a seguito di un attacco buffer overflow.

- Se il Virtual Stack List non è presente nel database o qualche return address è mancante, viene rilevata una Return Address Anomaly.

- Se il Virtual Stack List è presente nel database ma non è associato alla corretta system call, viene rilevata una Syscall Anomaly.

- Se il Virtual Path tra l'ultima system call e la corrente non è presente nel database, viene rilevata una Virtual Path Anomaly.

Capitolo 4

Implementazione

Il codice sorgente e/o i binari del modello presentato nel capito precedente non sono stati resi disponibili dagli autori, così per testarne le caratteristiche, le proprietà e i problemi è stato necessario reimplementare il modello. In questo capitolo vengono chiariti alcuni concetti alla base dei sistemi operativi e la loro implementazione nei sistemi GNU/L. La conoscenza approfondita di tali concetti è fondamentale per il lavoro di implementazione del modello teorico presentato nel capitolo precedente. Verranno inoltre proposte alcune soluzioni implementative per superare i problemi incontrati durante lo sviluppo.

4.1 System Call

In questo paragrafo verranno introdotti i concetti fondamentali sulle system call, in particolare verrà analizzata la ptrace e le system call la cui esecuzione può essere interrotta dall'arrivo di un segnale.

4.1.1 Concetti di base

I sistemi operativi offrono ai processi eseguiti in User Mode un insieme di interfacce per interagire con l'hardware e le periferiche come per esempio la CPU, i dischi, le stampanti ecc. I sistemi Unix implementano molte interfacce tra i processi User mode

e l'hardware sottostante, tra le quali le system call, che vengono poi inviate al kernel. Quando un processo in User Mode invoca una system call, la CPU passa al Kernel Mode ed esegue la funzione a livello kernel.
Per invocare una system call in linux deve essere eseguita l'istruzione in linguaggio assembly *int $0x80*, che richiama una eccezione programmata. Siccome il kernel implementa varie system call, il processo deve anche passare un parametro, il numero della syscall, utilizzato per identificare la system call desiderata; per questo scopo viene utilizzato il registro *eax*.
Tutte le system call ritornano un valore intero che indica il successo o meno dell'esecuzione. La convenzione per questi valori di ritorno sono diversi da quelle adottate per le *wrapper routine*, routine il cui unico scopo è quello di lanciare una syscall. Nel kernel, valori positivi o il valore 0 indicano una terminazione con successo della system call, mentre valori negativi indicano una condizione di errore. In quest'ultimo caso il valore ritornato è l'opposto del codice di errore che deve essere ritornato all'applicazione nella variabile *errno*. La variabile *errno* infatti non viene utilizzata in Kernel Mode ma solamente in User Mode.
Le *wrapper routine* invece hanno il compito di settare questa variabile quando la syscall è stata eseguita.
Il gestore delle system call (system call handler), che ha una struttura simile agli altri handler delle eccezioni, esegue le seguenti operazioni:

- Salva il contenuto di alcuni registri dello stack dell'applicazione in User Mode nello stack in Kernel Mode (questa operazione è comune a tutte le system call ed è codificata in linguaggio assembly).

- Esegue la system call invocando la corrispondente funzione C chiamata *system call service routine* in Kernel Mode.

- Esce dall'esecuzione della syscall attraverso la chiamata della funzione *ret_from_sys_call()* (che è codificata in linguaggio assembly) che riporta l'esecuzione in User Space

ripristinando l'esecuzione del processo che ha fatto la chiamata.

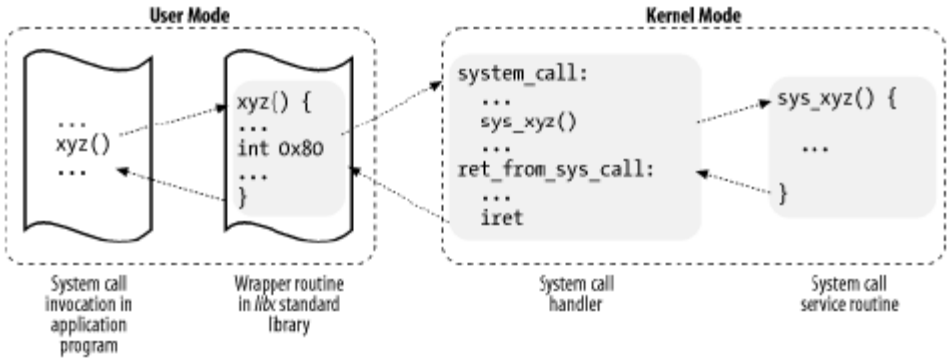

Figura 4.1: Chiamata di una system call.

Per associare ciascun numero delle system call con la corrispondente service routine, il kernel utilizza una system call dispatch table, salvata nell'array *sys_call_table* formato da *NR_syscalls* elementi (generalmente 256). L'elemento n-esimo contiene l'indirizzo della service routine della system call numero *n*. La macro *NR_syscalls* è un limite statico al numero massimo di system call implementabili; non indica il numero di syscall effettivamente implementate. Infatti la dispatch table può contenere tra le sue entry l'indirizzo della funzione *sys_ni_syscall( )*, che è una service routine delle syscall non implementate; e ritorna all'applicazione il codice d'errore *-ENOSYS*.

4.1.2 Ptrace

Per intercettare le system call si è utilizzato la syscall *ptrace* che permette ad un processo padre di osservare e controllare l'esecuzione di un altro processo, osservandone il record di attivazione compresi tutti i registri.
Questa syscall è principalmente usata nell'implementazione di breakpoint debugging e per tracciare system call come in questo caso.
Il processo padre comincia a tracciare un processo già esistente usando la chiamata *PTRACE_ATTACH*. Mentre è tracciato, il processo figlio si fermerà tutte le volte che

riceve un segnale, anche se il segnale viene ignorato (con l'eccezione del segnale *SIGKILL*, che mantiene il suo normale effetto). Il processo padre verrà notificato della condizione di stop del figlio alla prossima wait[1] e potrà ispezionarne lo stato. Il padre a questo punto può far continuare l'esecuzione del figlio ignorando il segnale che è stato mandato o mandandogli un segnale diverso.
Il comando *PTRACE_SYSCALL* ha l'effetto di bloccare il figlio ogni volta che invoca una syscall, dal punto di vista del processo padre sembrerà che il processo figlio sia stato stoppato dalla ricezione di una segnale di tipo *SIGTRAP*, più precisamente tale chiamata permette di bloccare il processo sia all'entrata di una syscall (syscall non ancora eseguita), sia all'uscita (syscall eseguita). Nella implementazione proposta il processo padre (il tracer) impone lo stop del processo figlio nella posizione di exit della system call.
Con il comando *PTRACE_GETREGS* si è in grado di copiare il contenuto dei registri del processo figlio tracciato in una zona dati del processo padre.
I registri a cui è possibile accedere sono raggruppati in una struct chiamata *user_regs_struct* (dichiarata in *<linux/user.h>*) dove è possibile trovare il numero della syscall, i parametri passati, i valori del program counter e del return address della funzione. Il numero della system call può essere trovato nel registro *eax*, il program counter nell'*eip*, il return address nei 4 byte prima del frame pointer *ebp*.

4.1.3 Syscall lente

Le richieste associate all'esecuzione di una syscall non possono essere sempre immediatamente soddisfatte dal kernel; quando questo accade, il processo che ha invocato la system call viene messo in uno stato *TASK_INTERRUPTIBLE* o *TASK_UNINTERRUPTIBLE*.
Se il processo è messo in uno stato *TASK_INTERRUPTIBLE* e qualche altro processo gli manda un segnale, il kernel mette il processo nello stato *TASK_RUNNING* senza completare l'esecuzione della syscall. Quando questo accade, visto che la syscall non ha completato la sua esecuzione, la system call service routine ritorna un error code

[1] La chiamata di sistema *wait()* è usata per attendere cambiamenti di stato in un figlio del processo chiamante, e ottenere informazioni sul figlio il cui stato è cambiato.

del tipo *EINTR, ERESTARTNOHAND, ERESTARTSYS, o ERESTARTNOINTR.*
In pratica l'unico codice di errore che un processo eseguito in User Mode può ottenere in questa situazione è *EINTR*, che quindi indica che la syscall non è stata completata. Gli altri codici di errore vengono usati internamente dal Kernel per sapere se la syscall deve essere rieseguita automaticamente o alla fine dell'esecuzione del signal handler. Nella Tabella 4.2 vengono evidenziate le relazioni tra i tipi di errore e le tre possibili azioni che possono essere compiute alla ricezione di un segnale circa la necessità di dover rieseguire la syscall o meno.

Azione	EINTR	ERESTARTSYS	ERESTARTNOHAND	ERESTARTNOINTR
Default	Termina	Riesegue	Riesegue	Riesegue
Ignorato	Termina	Riesegue	Riesegue	Riesegue
Gestito	Termina	Dipende	Termina	Riesegue

Tabella 4.1: Relazioni tra i tipi di errore e le possibili azioni compiute alla ricezione di un segnale.

I termini che appaiono nella tabella sono definiti nella seguente lista:

Termina La system call non sarà automaticamente rieseguita, il processo riprenderà l'esecuzione in User Mode dall'istruzione successiva alla *int $0x80* e il registro *eax* conterrà il valore -*EINTR*.

Riesegue Il kernel forza il processo in User Mode a riempire il registro *eax* con il numero della syscall e a rieseguire l'istruzione *int $0x80*; il processo non è consapevole della riesecuzione e non viene passato in User Mode nessun codice di errore.

Dipende La system call viene rieseguita solamente se viene settato il flag *SA_RESTART* del segnale inviato, diversamente la system call termina con il codice di errore -*EINTR*.

Quando arriva un segnale, il kernel deve essere sicuro che il processo stesse realmente eseguendo una syscall, prima di rieseguirla. In questo caso il registro *orig_eax* gioca un ruolo fondamentale.

Vediamo come questo registro viene inizializzato quando parte un interrupt o un exception hadler:

Interrupt Il campo contiene il numero IRQ associato all'interrupt meno 256.

0x80 exception Il campo contiene il numero della system call

Other exception Il campo contiene il valore -1

Per questi motivi un valore non negativo nel campo *orig_eax* significa che il segnale ha risvegliato un processo *TASK_INTERRUPTIBLE* che stava aspettando di finire una system call. La service routine riconosce così che la system call era stata interrotta, e quindi ritorna uno dei codici di errore visti in precedenza.
Se il segnale viene esplicitamente ignorato o se viene eseguita l'azione di default, la funzione do_signal() analizza il codice di errore della system call per decidere se la system call non terminata debba essere rieseguita. Nel caso in cui la debba rieseguire, la funzione modifica i registri in modo tale che ritornando in User Mode, *eip* punti all'istruzione *int $0x80* e *eax* contenga il numero della syscall.
Se invece il segnale è gestito, la funzione *hadle_signal()* analizza il codice di errore e il flag *SA_RESTART* della sigaction table per decidere la riesecuzione della syscall.
Se la syscall deve essere rieseguita la funzione *handle_signal* procede esattamente come la *do_signal()*; in caso contrario ritorna al processo in User Mode il codice di errore *-EINTR*.
Il prototipo implementato, analizzando i valori dei registri e i valori ritornati dalle funzioni, è in grado di sapere se l'esecuzione di una system call è stata interrotta e se questa verrà rieseguita. Grazie a queste informazioni si è in grado di riconoscere la riesecuzione di una syscall, non inserendo nelle strutture dati previste dal modello informazioni già inserite ed evitare così la costruzione di Virtual Path inesistenti.

4.2 Segnali

In questo paragrafo verranno introdotti i concetti fondamentali sui segnali, sui relativi problemi implementativi e sulle soluzioni adottate.

4.2.1 Concetti di base

Un segnale è un brevissimo messaggio che può essere mandato da un processo ad un gruppo di processi. L'unica informazione data al processo è tipicamente il numero che identifica il tipo di segnale.
I segnali vengono principalmente utilizzati per due scopi principali:

- per segnalare ad un processo che si è verificato un determinato evento
- per forzare un processo a eseguire una funzione di gestione del segnale (signal handler) di solito contenuta nel codice del processo

Ovviamente le due possibilità non sono mutualmente esclusive, visto che spesso un processo deve reagire agli eventi eseguendo una specifica routine.
Una importante caratteristica dei segnali è che possono essere inviati ad un processo in qualunque momento, il cui stato è imprevedibile.
Un processo può rispondere ad un segnale in tre modi:

1. ignorando esplicitamente il segnale.

2. eseguendo l'azione impostata di default dal kernel associata a quel segnale.

3. gestendo il segnale invocando la corrispondente funzione signal handler.

Una funzione di gestione del segnale (signal handler) è la funzione che viene eseguita alla ricezione del corrispondente segnale. Alla ricezione di un segnale gestito, il processo sospende la sua esecuzione ed esegue la funzione signal handler per poi riprendere l'esecuzione al ritorno del signal handler.
Se è stata definita una funzione signal handler per un determinato segnale, deve essere eseguita la funzione do_signal(), e questo avviene invocando handle_signal():

```
handle_signal(signr, ka, &info, oldset, regs);
return 1;
```

La funzione do_signal ritorna dopo aver gestito un singolo segnale; gli altri segnali in sospeso non vengono considerati fino alla prossima invocazione della do_signal(). Questo approccio assicura che i segnali real-time siano gestiti nell'ordine appropriato.
L'esecuzione di un signal handler è piuttosto complessa siccome è necessario passare dalla modalità di esecuzione User Mode a quella Kernel Mode.
Le funzioni signal handler vengono definite ed eseguite in User Mode dai processi. La funzione handle_signal invece viene eseguita in Kernel Mode; questo significa che il processo deve prima eseguire il signal handler in User Mode prima di poter tornare alla normale esecuzione del processo. Inoltre, quando il kernel tenta di riprendere la normale esecuzione del processo, lo stack in Kernel Mode non contiene più il contesto hardware del programma interrotto perché lo stack in Kernel Mode viene svuotato ad ogni passaggio da User Mode a Kernel Mode.
Un'ulteriore complicazione è data dal fatto che il signal handler potrebbe invocare a sua volta delle syscall. In questo caso, una volta eseguite le service routine, il controllo dell'esecuzione deve ritornare al signal handler invece che al codice del programma interrotto.
La soluzione adottata in Linux consiste nel copiare il contesto hardware salvato nello stack Kernel Mode, nello stack User Mode del processo corrente. Lo stack User Mode viene modificato in maniera tale che quando il signal handler termina, venga automaticamente chiamata la syscall sigreturn(), che copia il contesto hardware nello stack Kernel Mode e ripristina il contenuto originario dello stack User Mode.
A questo punto è necessario vedere in dettaglio cosa accade nel momento in cui viene mandato un segnale ad un processo.
Per gestire un segnale, un interrupt o una exception il processo passa in Kernel Mode. Prima di ritornare in User Mode, il kernel esegue la funzione do_signal() che gestisce il segnale (invocando la handle_signal()) e modifica lo stack User Mode (invocando setup_frame() o setup_rt_frame()). Quando il processo ritorna in User Mode, viene forzata l'esecuzione del signal handler mettendo l'indirizzo dell'handler nel program counter. Quando questa funzione termina, viene eseguito il codice di ritorno messo

sullo stack User Mode dalle funzioni setup_frame() o setup_rt_frame(). Questo codice invoca la system call sigreturn(), la cui service routine copia il contesto hardware del processo nello stack Kernel Mode e ripristina lo stack User Mode riportandolo allo stato originale (invocando restore_sigcontext()). Quando la system call termina il programma riprende la sua normale esecuzione.

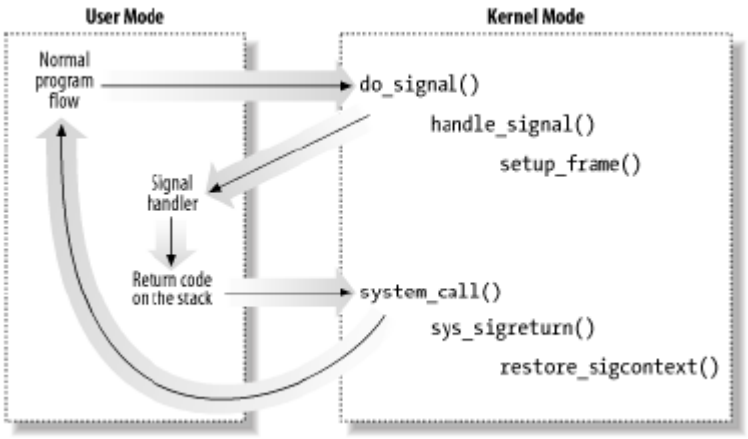

Figura 4.2: Gestione di un segnale.

4.2.2 Problemi e Soluzioni implementative

Il processo con il compito di monitorare l'applicazione protetta, chiamato tracer deve essere in grado di isolare tutti quei comportamenti del programma che dipendono o sono causati da eventi casuali o comunque impredicibili. Se così non facesse, le tracce generate conterrebbero oltre che syscall non previste dall'applicazione, anche Virtual Path inesistenti.

Supponiamo che l'applicazione preveda la gestione e quindi la relativa funzione handler del segnale 12 (SIGUSR2), alla ricezione di questo segnale l'applicazione viene interrotta dal kernel che passa l'esecuzione all'handler del segnale 12. Il tracer in questo caso registrerebbe tutte le syscall eseguite dall'handler come se fossero state eseguite dall'applicazione e creerebbe un Virtual Path tra l'ultima syscall eseguita dall'applicazione e la prima syscall eseguita dall'handler.

Questo ovviamente è sbagliato, in quanto il momento (che per noi significa la posizione nello stack) in cui si è passati ad eseguire il signal handler è totalmente casuale e in una successiva fase di detection, l'esecuzione dell'applicazione senza l'arrivo del segnale verrebbe considerata come un'anomalia.
Per questo motivo sarebbe necessario distinguere l'esecuzione dell'applicazione dalle gestioni dei segnali.
Data l'impredicibilità dell'arrivo dei segnali appunto, il tracer dovrà essere in grado di fermare la sua esecuzione e riprenderla non appena la gestione del segnale, se gestito, è terminata. Se il segnale non è gestito nessuna funzione verrà invocata quindi non verranno eseguite syscall e non ci saranno cambiamenti nello stack.
La system call signal() viene usata per settare un signal handler per un unico tipo di segnale. La funzione signal() accetta due parametri, il numero del segnale che deve gestire e un puntatore alla funzione signal handler associata. In questo modo, ogni volta che un segnale ha un signal handler associato, il processo chiama la funzione signal() con il numero e il puntatore associati.
Grazie al dynamic linker[2] è stato possibile scrivere una libreria che sostituisca la syscall signal() con un'altra funzione signal() (un wrapper della syscall signal()) che scrive su di un file il primo parametro passato alla funzione, cioè il numero del segnale gestito. Linux prevede l'utilizzo di librerie condivise e in questo caso gran parte del lavoro di associazione dei nomi a locazioni, che tradizionalmente era svolto al momento del link, deve essere ritardato al momento del load (caricamento).
Esistono diverse variabili di ambiente che influenzano il comportamento del loader dinamico tra queste LD_PRELOAD. LD_PRELOAD può essere impostata in modo tale da caricare un file contenente delle definizioni di funzioni da sovrapporre alle esistenti. Nel nostro caso vogliamo sostituire la funzione signal() con una routine sostitutiva; possiamo compilare la libreria e creare il file miasignal.so da utilizzare con i comandi

```
LD_PRELOAD=miasignal.so; export LD_PRELOAD
```

[2] il dynamic linker è quella parte del sistema operativo che esegue le operazioni di load e link delle librerie condivise.

Tutte le volte che viene lanciato l'eseguibile dell'applicazione da tracciare, verrà caricata la libreria miasignal.so, in questo modo si è in grado di sapere tutti i segnali gestiti dall'applicazione.
A questo punto il mio tracer dovrà fermarsi ogni volta che riceverà un segnale che è gestito e riprendere l'esecuzione al ritorno del signal handler, cioè quando verrà eseguita la syscall sigreturn, come spiegato nel paragrafo precedente.

4.3 Informazioni dello Stack

Il call stack è uno speciale tipo di stack nel quale vengono salvate le informazioni delle subroutine attive[3] di un programma.
Questo tipo di stack viene anche chiamato execution stack, control stack, function stack, e spesso abbreviato in "stack". Il call stack viene utilizzato per vari scopi, ma principalmente per sapere in che punto ciascuna subroutine deve ritornare quando ha completato la propria esecuzione.

4.3.1 Concetti di base

Caratteristica dei linguaggi ad alto livello è la capacità di poter definire procedure o funzioni. Le procedure possono ricevere e restituire parametri al chiamante e definire variabili locali. Quando una routine ha termine si deve ripristinare il corretto flusso del programma ritornando all'istruzione immediatamente successiva alla chiamata a funzione: per compiere queste operazioni abbiamo bisogno dello stack.
Lo stack è un insieme di blocchi contigui di memoria ed ha una struttura di tipo LIFO (Last In First Out). Nelle architetture x86 lo stack cresce verso il basso quindi verso indirizzi di memoria numericamente minori.
Il termine stack o pila viene usato in diversi contesti per riferirsi a strutture dati le cui modalità d'accesso seguono una politica LIFO, ovvero tale per cui i dati vengono estratti (letti) in ordine rigorosamente inverso rispetto a quello in cui sono stati inseriti

[3]Le subroutine attive sono quelle routine che sono state chiamate ma che non hanno ancora completato la loro esecuzione.

(scritti).
Gli stack sono implementati usando funzioni predefinite e scritte per facilitarne l'utilizzo. Le principali operazioni sono push e pop; push aggiunge un elemento in cima alla lista, pop prende un elemento dalla cime della lista.
Nelle architetture x86 c'è un registro dedicato che contiene l'indirizzo dell'ultima locazione di memoria occupata sullo stack, esso è il registro ESP che prende il nome di stack pointer. Teoricamente si potrebbe specificare la locazione di memoria di ciascuna variabile con spiazzamenti relativi allo stack pointer, questo è molto scomodo perché lo stack viene continuamente allocato e deallocato, quindi gli indirizzamenti relativi allo stack pointer cambierebbero in continuazione. Per ovviare a questo problema viene usato un altro registro per tenere traccia della prima locazione di memoria del record di attivazione di una procedura, esso è il registro EBP che prende il nome di frame pointer o base pointer.
Specificare gli indirizzi delle variabili locali ad una procedura rispetto al frame pointer risulta conveniente poiché il suo valore rimane invariato nell'arco della vita di una procedura. La Figura 3.3 mostra la posizione puntata dai registri EBP e ESP nel record

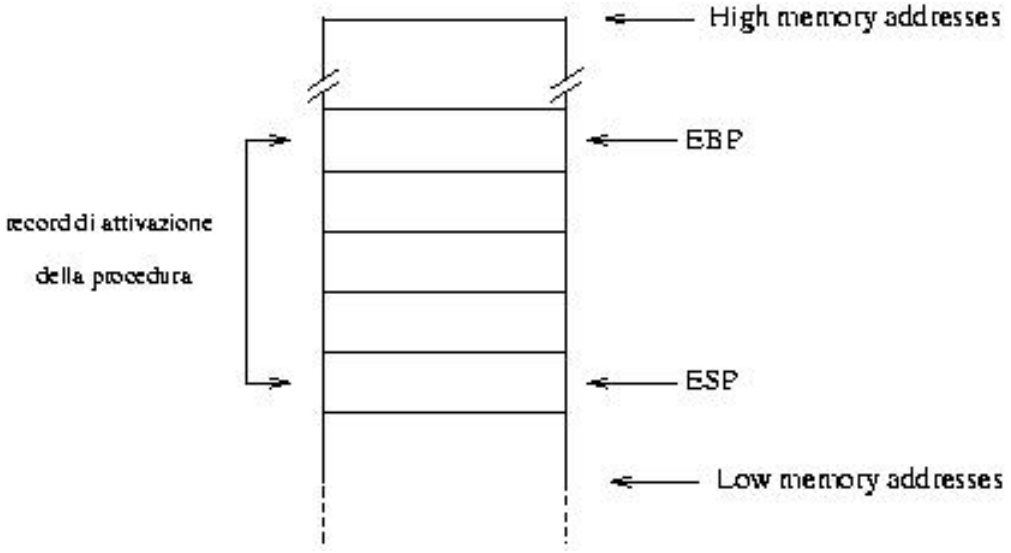

Figura 4.3: Record di attivazione di una funzione.

di attivazione di una generica procedura. Si noti che l'indirizzo contenuto all'interno del registro ESP è minore rispetto a quello in EBP.

Dopo una push il valore dello stack pointer risulterà numericamente minore rispetto alla locazione dove puntava in precedenza. Al contrario quando si esegue un'istruzione pop il valore dello stack pointer verrà incrementato.

4.3.2 Prologo

Analizzando una generica funzione con l'aiuto di gdb è possibile vedere cosa avviene subito dopo la sua chiamata:

```
(gdb) disassemble esempio
Dump of assembler code for function esempio:

0x80483b4 <funzione>:       push   %ebp
0x80483b5 <funzione+1>:     mov    %esp,%ebp
0x80483b7 <funzione+3>:     sub    $0x18,%esp
```

Le prime tre istruzioni, definite prologo della procedura, sono:

```
1 push   %ebp
2 mov    %esp,\%ebp
3 sub    $0x18,%esp
```

Viene salvato sullo stack il valore del registro EBP (istruzione 1) e viene messo il valore di ESP in EBP (istruzione 2). Successivamente viene sottratto 24 allo stack pointer per allocare lo spazio necessario per le variabili locali (istruzione 3).
Prima della chiamata alla procedura EBP ed ESP puntano rispettivamente alla base ed all'ultima locazione di memoria occupata dalla procedura chiamante. Quando la procedura chiamata effettua la push del registro EBP sullo stack, lo stack pointer punterà alla locazione di memoria dove è stato salvato il base pointer della procedura chiamante. Quest'ultima locazione di memoria d'ora in avanti sarà chiamata saved frame pointer (SFP).
A questo punto la procedura chiamata copia lo stack pointer nel suo frame pointer, in questo modo il registo EBP punta alla prima locazione di memoria del suo record

di attivazione. Successivamente alloca memoria per le variabili locali sottraendo lo spazio necessario al valore dello stack pointer.

4.3.3 Chiamata a procedura

Ricordiamo che esiste un registro che punta all'area di memoria dove si trova l'istruzione successiva rispetto a quella in esecuzione. Questo è il registro EIP, che prende il nome di instruction pointer.
In assembly una chiamata a funzione si traduce con l'istruzione call, essa ha due compiti fondamentali:

- alterare il normale flusso del programma facendo eseguire come istruzione successiva la prima istruzione della procedura chiamata
- salvare sullo stack l'indirizzo dell'istruzione successiva ad essa nel chiamante

Quando la procedura chiamata avrà termine si dovrà tornare ad eseguire l'istruzione successiva alla call nel chiamante, per questo motivo viene salvato sullo stack il suo indirizzo. Chiameremo questa cella di memoria saved return pointer (SRET).
Osserviamo come si presenta lo stack dopo l'esecuzione del prologo di una generica funzione f in Figura 4.4. Esso conterrà le variabili locali al chiamante, il saved return pointer, il saved frame pointer e successivamente le variabili locali alla funzione f.

4.3.4 Epilogo

Le ultime due istruzioni di una procedura, definite epilogo, sono:

```
leave
ret
```

L'istruzione leave ha il compito di ripristinare i registri EBP ed ESP in modo che essi tornino a puntare rispettivamente la prima e l'ultima locazione di memoria occupate sullo stack dalla procedura chiamante.
L'istruzione ret ha il compito di indirizzare la CPU ad eseguire le istruzioni successive

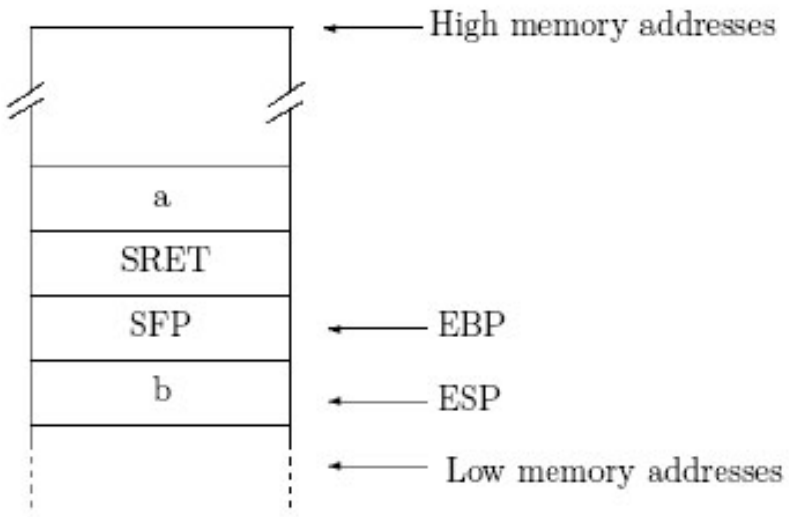

Figura 4.4: Stack dopo l'esecuzione del prologo di una generica funzione f.

alla call nel chiamante. Questa istruzione "copierà" il contenuto di SRET all'interno di EIP, così facendo la CPU eseguirà l'istruzione puntata da SRET che è quella successiva alla call all'interno della procedura chiamante.
Vediamo un esempio in pseudo assembly per chiarire questo passo cruciale:

```
        somma:
0x0001      add $5,%ecx
0x0002            subl $5,%ecx
0x0003            leave
0x0004            ret
        main:
0x0005      add $5,%eax
0x0006      call somma
0x0007      add $4,%eax
0x0008      leave
0x0009      ret
```

Sulla sinistra è stato messo un ipotetico indirizzo di memoria dove si trova l'istruzione, supponendo che tutte le istruzioni occupino la stessa dimensione.

Quando all'interno del main viene chiamata la funzione somma, l'istruzione call salva sullo stack, in SRET, l'indirizzo 0x0007. Ora si passa ad eseguire la funzione somma che eseguirà tutte le sue istruzioni sino a ret. Ret copierà il contenuto di SRET in EIP, così facendo la CPU eseguirà l'istruzione 0x0007. In questo modo abbiamo ripristinato il flusso logico del programma tornando ad eseguire le istruzioni che seguono la call all'interno del main.

4.3.5 Stack walking

Il modello presentato nel capitolo precedente, ogni volta che viene eseguita una system call, per la costruzione dei Virtual Stack List e dei Virtual Path utilizza gli indirizzi di ritorno estratti dallo stack di tutte le funzioni non ancora ritornate.
Per estrarre tutti i return address dello stack è necessario percorrere a ritroso lo stack di esecuzione del processo tracciato. Questo tipo di operazione viene chiamata stack walking. Partendo dal record di attivazione della syscall appena eseguita, si estrae il valore del registro *EBP*, che punta al registro *SFP*, necessario per poi saltare nel record di attivazione da dove è stata chiamata la system call. Inoltre si estrae il valore contenuto nei 4 byte precedenti alla locazione del registro *EBP*, cioè il valore del *RET*, l'indirizzo dove saltare al ritorno della funzione. Estraendo i valori dei registri *EBP* e seguendo gli indirizzi puntati dagli *SFP* si giunge alla cima dello stack, collezionando tutti i valori degli indirizzi di ritorno *RET*. Viene ora riportato un esempio dell'esecuzione dello stack walking di un semplice programma.

```
0x0001 int somma()
0x0002 {
0x0003 ...
0x0004 }
0x0005 int main()
0x0006 {
0x0007 somma;
0x0008 }
```

Nella Figura 4.5, viene riportato lo stato dello stack al momento dell'esecuzione della system call read(). Il tracer tracciando l'esecuzione del programma si fermerà all'esecuzione di ogni syscall ed estrarrà il valore *RET*, di ogni record di attivazione presente sullo stack partendo da quello della syscall fino ad arrivare alla cima dello stack.

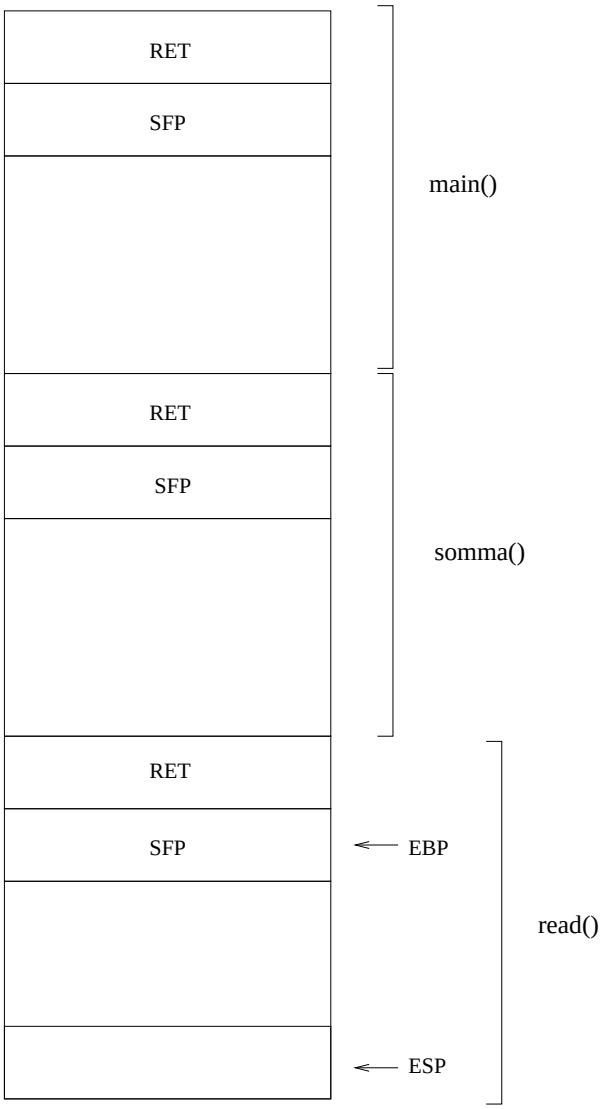

Figura 4.5: Contenuto dello stack durante l'esecuzione della read().

4.3.6 Problemi dello Stack walking

Come spiegato in un paragrafo precedente il prologo all'inizio di ciascuna funzione è utile in quanto è possibile utilizzare EBP, che rimane fisso, per calcolare gli offset dei parametri. Sfortunatamente non tutti i programmi o le librerie vengono compilati con il prologo[4], e senza il prologo lo stack walking risulta incompleto. Non è più possibile distinguere le chiamate di due funzioni dello stesso tipo effettuate in due punti diversi del programma.
Dagli esperimenti fatti si è notato che anche le routine di alcune system call non fanno il prologo e visto che ovviamente non è sempre possibile ricompilare tutte le librerie e le applicazioni che si vogliono tracciare, si è cercata una soluzione trasparente al programma tracciato.
Si consideri l'esecuzione del seguente programma:

```
int somma()
{
read(0,var1,4);
write(1, "a1", 2);
read(0,var2,4);
write(1, "a2", 2);
}
int main()
{
somma;
}
```

In Figura 4.6 viene riportato lo stato dello stack al momento dell'esecuzione della prima e della seconda syscall read(), che come ricordato prima, sono system call che non eseguono il prologo; e i relativi Virtual Stack List.
In questo caso il tracer non sarebbe in grado di distinguere le due system call read().

[4]In GCC con l'opzione -fno-omit-frame-pointer è possibile compilare le funzioni senza il prologo, in questo caso viene utilizzato il registro ESP al posto di EBP.

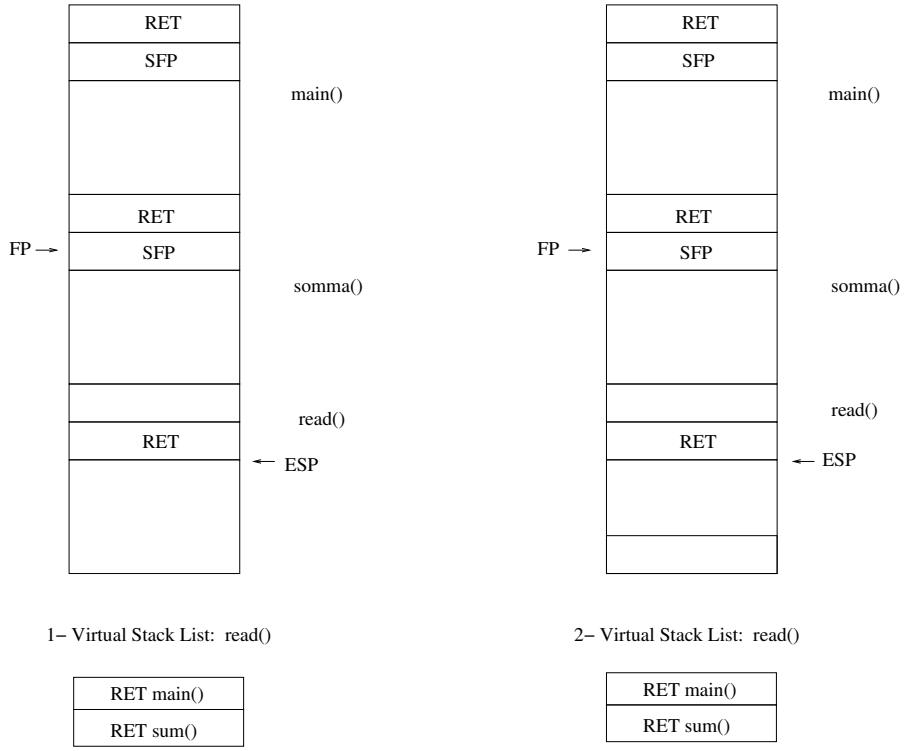

Figura 4.6: Stato dello stack all'esecuzione della prima e della seconda read().

Come già spiegato nel paragrafo riguardante i segnali esistono diverse variabili di ambiente che influenzano il comportamento del loader dinamico, tra queste LD_PRELOAD. Attraverso la variabile LD_PRELOAD può essere caricata una libreria contenente le definizioni delle funzioni da sovrapporre alle esistenti. Creando una libreria con la ridefinizione di tutte le routine chiamate dalle syscall è stato possibile per ogni syscall aggiungere sullo stack il valore del RET, così da permettere la corretta esecuzione dello stack walking.
In figura 4.7 è possibile vedere lo stack dell'esecuzione delle due syscall read() e dei relativi Virtual Stack List dopo aver caricato la libreria. I Virtual Stack List delle due system call sono diversi ed è quindi possibile distinguere le due esecuzioni.

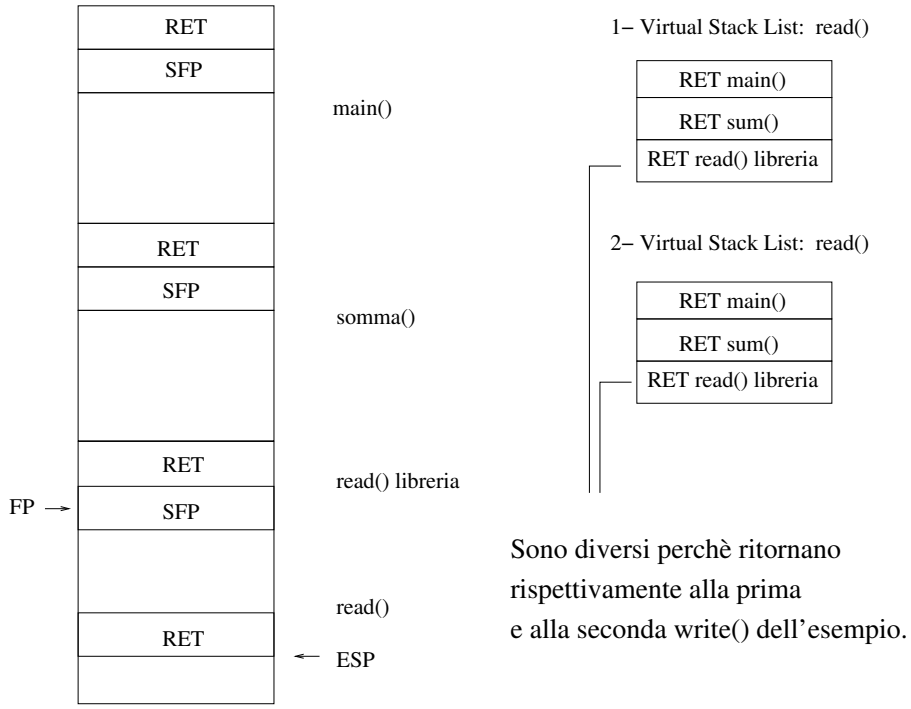

Figura 4.7: Stato dello stack all'esecuzione della prima e della seconda read() con la libreria caricata.

4.4 Gestione processi figli

Come spiegato nei paragrafi precedenti il tracer implementato utilizza la syscall ptrace per tracciare l'esecuzione del processo da monitorare. Il modello teorico proposto [7] non prevede una soluzione ne teorica ne implementativa circa la possibilità di tracciare anche i processi figli creati dal processo monitorato.
Durante la fase di implementazione del modello si è cercata una soluzione a questo problema.
Analizzando le syscall eseguite dal processo si è in grado di sapere il momento in cui

viene eseguita una syscall fork[5] o una execve[6] e il PID dei processi creati analizzando il valore ritornato dalle stesse syscall.
Il tracer implementato, quando rileva l'esecuzione di una syscall fork o execve, esegue lui stesso una fork per poter seguire e tracciare l'esecuzione di tutti i figli. Le informazioni raccolte, cioè i Virtual Stack List e i Virtual Path di ogni figlio tracciato, vengono salvati in database separati, e solo alla fine della fase di learning verrà fatta una normalizzazione delle informazioni raccolte per poter eliminare i database contenenti comportamenti uguali.
In fase di detection il detector, tutte le volte che rileverà l'esecuzione di una fork o di una execve, dovrà eseguire lui stesso una fork e seguire il figlio creato confrontando le informazioni raccolte in real-time, Virtual Stack List e Virtual Path, con quelle salvate in tutti i database riguardanti i figli.
Il modello teorico del VTPath infatti non prevede la possibilità in fase di detection di distinguere in qualche maniera i processi figli, se non dal punto in cui viene eseguita la syscall fork o execve nel processo padre. Per questo motivo il detector associato ad un processo figlio deve seguire tutti i database associati ai figli creati nello stesso punto dell'applicazione (stesso call site della fork o execve), e non rilevare un'anomalia fino al momento in cui nessun database conterrà le syscall,Virtual Stack List, Virtual Path salvati.
Come è facile immaginare la gestione di figli multipli risulta onerosa in termini di performance, infatti si tenga presente che viene lanciata una istanza del tracer per ogni fork o execve eseguita. La situazione peggiora in fase di detection in cui ogni istanza del detector deve confrontare i dati raccolti in real-time con tutti quelli contenuti nei database dei figli creati nella stessa posizione dell'applicazione.

[5] fork() crea un processo figlio che differisce dal processo padre solo nei suoi PID e PPID, e nel fatto che l'utilizzo delle risorse è impostato a 0. Sotto Linux fork() è implementata usando pagine copy-on-write (i dati vengono copiati solo al momento della loro effettiva modifica), in modo che l'unica penalizzazione in cui incorre è il tempo e memoria necessari per duplicare le tabelle delle pagine del genitore, e per creare una struttura di task unica per il figlio.

[6] execve() esegue il file specificato, trasformando il processo che la chiama in un nuovo processo. L'immagine del nuovo processo viene costruita a partire dal file oggetto eseguibile specificato e sovrapposta a quella del processo che invoca la execve(). Viene sovrapposto sia lo spazio codice che lo spazio dati. Il nuovo processo così ottenuto viene eseguito e poi esce e non vi è alcun ritorno al processo originale che è stato sovrapposto.

4.5 Database

Come spiegato nel capitolo 3, durante la fase di learning il tracer, per ogni system call eseguita, recupera le lista degli indirizzi di ritorno per costruire il Virtual Stack List. Questi indirizzi vengono memorizzati insieme al numero della syscall e al program counter nel database dei Virtual Stack List associato al processo tracciato.
Anche i Virtual Path che vengono costruiti sono salvati in una forma compatta in una tabella del database associato al processo.
Il prototipo implementato prevede anche la gestione dei segnali, quindi l'interruzione della fase di detection ed il relativo salvataggio delle informazioni dell'applicazione durante l'esecuzione del signal handler, e la successiva ripresa nel momento in cui il gestore termina la sua esecuzione.
Siccome in queste fasi in cui vengono eseguite le funzioni di gestione dei segnali l'applicazione potrebbe subire attacchi che non verrebbero rilevati, si è pensato, anziché interrompere il tracciamento e il salvataggio delle informazioni, semplicemente di salvare le informazioni in database separati in base al tipo di segnale gestito.
In questo modo per ogni processo tracciato verrà creato un database contenente i Virtual Stack List e i Virtual Path, e un database contenente la lista dei segnali gestiti e i relativi Virtual Stack List e i Virtual Path eseguiti dalle funzioni di gestione. Se il processo tracciato effettua delle syscall fork o execve, cioè crea dei nuovi processi, verrà creata una nuova istanza del tracer e i relativi database del processo e dei segnali gestiti.
La struttura dei database creati per ogni processo è la seguente:

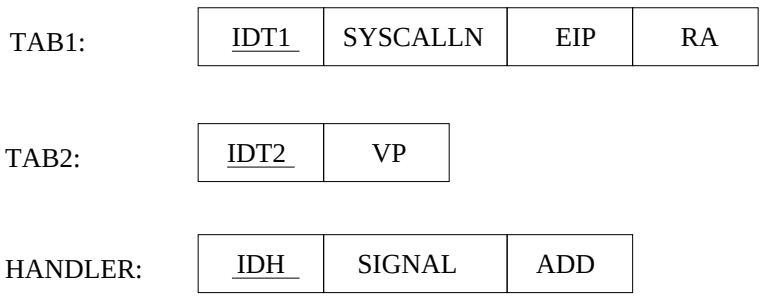

Figura 4.8: Tabelle dei Database.

Nella prima tabella vengono salvate le informazioni relative ai Virtual Stack list; il campo ID rappresenta la chiave primaria, il campo SYSCALL conterrà il numero della syscall tracciata, il campo EIP l'indirizzo dello stack da dove è stata chiamata la syscall (call site), il campo RA la stringa di tutti gli indirizzi di ritorno raccolta durante la fase di stack walking (ogni indirizzo è separato dall'altro con un carattere speciale).
Nella seconda tabella vengono salvate le informazioni relative ai Virtual Path; il campo ID rappresenta la chiave primaria, il campo VP la stringa di tutti gli indirizzi contenuti nella struttura dati Virtual Path definita nel capitolo 3 (ogni indirizzo è separato dall'altro con un carattere speciale).
Nella tabella Hand contenuta nel database di gestione dei segnali, sono contenuti il campo ID che rappresenta la chiave primaria, il campo SIGNAL contenente il numero del segnale gestito, il campo ADD l'indirizzo della funzione di gestione del segnale.

Per gestire i database nel prototipo implementato è stata utilizzato SQLite [2].
SQLite è una libreria C che implementa un DBMS SQL incorporabile all'interno di applicazioni. Il suo creatore è D. Richard Hipp, che lo ha rilasciato come software Open Source di pubblico dominio, privo di qualsiasi licenza. Essendo una libreria, non è un processo standalone utilizzabile di per sé, ma può essere linkato all'interno di un altro programma. Il pacchetto ha molte interessanti caratteristiche: è molto piccolo (meno di 250KB per l'intera libreria), ed è molto veloce; in molti casi più veloce di MySQL e PostgreSQL.

4.6 Architettura dell'IDS

In questo paragrafo viene presentata l'architettura dell'IDS implementato. I concetti fondamentali per la comprensione dei dettagli implementativi del modello sono stati dati nei paragrafi precedenti.
Il processo avviene in due fasi principali: learning e detection. Nella fase di learning il tracer monitora l'esecuzione di un processo salvando nei database associati le informazioni raccolte. In figura viene riportato l'algoritmo ad alto livello del funzionamento del tracer.

Il tracer sfruttando le funzionalità offerte dalla system call ptrace è in grado di controllare l'esecuzione dell'applicazione monitorata, in particolare vengono sfruttate nelle funzioni ad alto livello alle righe 1,7,11.
Come già detto il tracer è in grado di gestire i segnali: alle righe 4,5,6 vediamo le pseudo funzioni per la ricezione del segnale e, se il segnale è gestito e la funzione associata esegue syscall, abbiamo la relativa costruzione delle strutture dati e il salvataggio nel database dei segnali. Alla fine dell'esecuzione della funzione di gestione del segnale, per ripristinare lo stack, come spiegato in precedenza, viene eseguita la system call sigreturn. Alla ricezione di questa system call il tracer ritorna a salvare le syscall e le strutture dati costruite nel database del processo.
Se il tracer si trova a tracciare una system call che ha già eseguito l'istruzione *int 0x80*, cioè è nella fase di exit, e non è stata interrotta, viene effettuato lo stack walking.
Con le informazioni raccolte dallo stack vengono costruite le strutture Virtual Stack List e Virtual Path e se non ancora presenti vengono salvate nel database appropriato.
Nella fase di detection il detector monitora l'esecuzione del processo confrontando le informazioni raccolte con quelle salvate nei database associati e se non sono presenti segnala un'anomalia.
Le operazioni eseguite dal detector sono simili a quelle eseguite dal tracer, solamente che il detector invece di inserire le informazioni nel database, esegue query (righe 7,23,25) e confronti e in caso di discrepanza segnala anomalie (8,20,24,26).

```
1:        pid=attach-process;
2:        while(1)
3:        {
4:                signal=get-signal;
5:                if (signal-is-handled)
6:                        {db=signal-db;}
7:                syscall=wait-process-on-syscall(pid);
8:                if (syscall not in db)
9:                        {save-SYSCALL-on-db;}
10:                get-register(pid);
11:                if(syscall=fork or syscall=execve)
12:                        {fork-tracer-to-child;}
13:                if(syscall=sigreturn)
14:                        {db=process-db;}
15:                if(exit-syscall)
16:                {
17:                if(!slow-syscall)
18:                        {
19:                        do-stack-walking;
20:                        VSL=create-Virtual-Stack-List;
21:                        VP=create-Virtual-Path;
22:                        if (VSL not in db)
23:                                {save-VSL-on-db;}
24:                        if (VP not in db)
25:                                {save-VP-on-db;}
26:                        }
27:                }
28:         }
```

Figura 4.9: Tracer

```
1.        pid=attach-process;
2.        while(1)
3.        {
4.                signal=get-signal;
5.                if (signal-is-handled)
6.                        {db=signal-db;}
6.                syscall=wait-process-on-syscall(pid);
7.                if (syscall not in db)
8.                         {SYSCALL-ANOMALY;}
9.                get-register(pid);
10.                if(syscall=fork or syscall=execve)
11.                        {fork-tracer-to-child;}
12.                if(syscall=sigreturn)
13.                        {db=process-db;}
14.                if(exit-syscall)
15.                {
16.                if(!slow-syscall)
17.                        {
18.                        error=do-stack-walking;
19.                        if (error)
20.                                {STACK-ANOMALY;}
21.                        VSL=create-Virtual-Stack-List;
22.                        VP=create-Virtual-Path;
23.                        if (VSL not in db)
24.                                {VSL-ANOMALY;}
25.                        if (VP not in db)
26.                                {VP-ANOMALY;}
27.                        }
28.                }
29.         }
```

Figura 4.10: Detector

Capitolo 5

Risultati Sperimentali

5.1 Efficacia del Modello

Come anticipato nel capitolo 2, alcuni modelli di IDS non sono in grado di rilevare alcuni tipi di attacchi chiamati IPE e Mimicry. Una vulnerabilità in una ben determinata posizione di un programma se adeguatamente sfruttata porterebbe l'attaccante a poter modificare il flusso di esecuzione del programma facendolo tipicamente saltare ad una posizione successiva ad un controllo di sicurezza, che verrebbe così evitato. Gli attacchi che sfruttano in questo modo certe vulnerabilità vengono chiamati Impossible Path Execution (IPE).

5.1.1 IPE:Attacco 1

L'attacco presentato in Figura 5.1 rappresenta il primo modello di attacco di tipo IPE implementato.
Considerando la funzione *login_user()*, è facile individuare due possibili percorsi di esecuzione data la presenza del costrutto *if()*. Se la funzione *is_regular(user)* ritorna *true* verrà seguito un percorso di esecuzione (I) altrimenti verrà seguito l'altro (II). Si supponga che la funzione *read_next_command()* chiamata alla riga 6 contenga una funzione *strcpy()* vulnerabile e un attaccante sia in grado di sostituire il return address

della funzione in modo tale che la *read_next_command()* ritorni alla riga 21 (II), cioè l'indirizzo dove sarebbe ritornata se avesse scelto l'altro percorso.
Nessuno dei modelli presentati nel capitolo 2 è in grado di rilevare questo tipo di attacco tranne il VTPath, siccome questo modello è in grado di distinguere tra la *sys_write()* chiamata seguendo il percorso I e quella chiamata seguendo il percorso II.
Data l'imprecisione delle informazioni raccolte dagli altri modelli di IDS una volta eseguita l'istruzione *jump* tali modelli non sono in grado di rilevare l'anomalia. Per l'IDS il percorso seguito infatti risulta essere il secondo.
Il modello VTPath è in grado di rilevare l'attacco siccome oltre a verificare i program counter e le transizioni di stato, tiene conto anche dei record di attivazione di entrambe le invocazioni della funzione *read_next_command()*. Più precisamente è in grado di rilevare un Virtual Path non valido dalla *sys_umask* alla *sys_write* nella funzione *read_next_command()* (percorso I), e rilevare che il return address della *read_next_command()* è cambiato dato l'overflow nella funzione *strcpy()*.

5.1.2 IPE:Attacco 2

L'attacco presentato in figura 5.2 rappresenta un altro tipico esempio di attacco IPE.
La funzione *f()* viene richiamata due volte nel *main()* per le seguenti due operazioni: controllare lo username e controllare la password. La funzione *f()* sceglie che operazione eseguire basandosi sul parametro che gli viene passato.
Tale parametro viene salvato nella variabile mode; se la variabile viene cambiata da un attaccante sfruttando l'overflow del buffer *input* adiacente, e viene inserito uno username valido e il resto viene riempito con degli zero, quando verrà chiamata la funzione f(1), il valore della variabile mode verrà cambiato in zero. In questo modo, invece di controllare prima lo username e poi la password, verrà controllato due volte lo username. In questo modo un attaccante può ottenere l'accesso senza conoscere la password.
Il modello del VTPath rileva questo attacco siccome viene calcolato un Virtual Path non valido tra la *sys_close* quando viene invocata la funzione *f(1)* e la successiva *sys_write* nel *main()*.
I modelli presentati nel capitolo precedente non sono in grado di rilevare questo attac-

```
1:void read_next_cmd() {
2:        uchar input_buf
3:        umask(2); // sys_umask()
4:        ...
5:        // copy a command
6:        strcpy( &input_buf[0], getenv("USERCMD"));
7:        printf( "\n" ); // sys_write()
8:}
9:void login_user(int user){
10:        if( is_regular(user)){
11:                // unprivileged mode
12:        read_next_cmd();// (I),this function will
13:                        // be overflowed
14:        ...
15:        // handle commands allowed to a regular user
16:        return;
17:}
18:        // privileged mode
19:        read_next_cmd();// (II), this function call
20:                        // will be skipped
21:        // --> this is where the control will be
22:        // transferred after a ret in read_next_cmd() at (I)
23:        seteuid(0);
24:        system( "rsync /etc/master.passwd ok@aeou.com:/ipe" );
25:        // and other privileged commands accessible only to
26:        // superuser
27:}
```

Figura 5.1: Pseudo codice attacco 1

co perché entrambe i rami nella funzione *f()* eseguono le stesse system call quindi la sequenza di syscall rimane la stessa durante l'attacco.
Il modello FSA non è in grado di rilevare l'attacco siccome il passaggio tra la funzione *sys_close()* e la *sys_write* risulta essere una transizione valida dell'FSA.

5.1.3 Generalizzazioni degli attacchi IPE

Basandosi sui due attacchi appena descritti è possibile fare le seguenti osservazioni generali. Per prima cosa in entrambi i casi è necessario avere la possibilità di cambiare il flusso di esecuzione del programma, cioè avere vulnerabilità ben definite; nei nostri

```
1:f(int arg){
2:        int mode = arg; // this variable is overflowed
3:        char input[10];
4:        fopen(); // sys_open(), open passwd file
5:        // overflow, changes mode variable => execution flow
6:        scanf("\%s", &input[0] );
7:        if( mode == CHECK_UNAME ){ // check username
8:                fread(); // sys_read(), read from passwd file
9:                fclose(); // sys_close()
10:                if(is_valid_user(input)) ret=1; else ret=0;
11:        }
12:        else if( mode == CHECK_PASSWD ){ // check password?
13:        fread(); // sys_read(), read from passwd file
14:        fclose(); // sys_close()
15:        if(is_valid_pass(input)) ret=1; else ret=0;
16:        }
17:        return ret;
18:}
19:void main(){
20:        printf( prompt ); // sys_write()
21:        ret=f(0); // (I), read/check username
22:        if( ret ) ret = f(1);//(II), read/check password
23:                                //if username was correct
24:        printf( "Authenticated\n" ); // sys_write()
25:        if( ret )
26:        execve( "/bin/sh" ); // superuser mode
27:}
```

Figura 5.2: Pseudo codice attacco 2

esempi si sono utilizzate le vulnerabilità di tipo buffer overflow. Considerando che i buffer overflow non sono sempre possibili crediamo comunque che la scelta sia giustificata visto che due terzi delle advisories del CERT [3] negli ultimi anni riguardavano buffer overflow [14].

I programmi che sono vulnerabili devono anche avere una determinata struttura che preveda, per esempio, una sezione critica dove poter saltare.

Negli attacchi descritti nei paragrafi precedenti sono stati presentati due esempi di possibili strutture che possono essere sfruttate per questi tipi di attacchi: un *if()* che gestisce una scelta “security-critical”; e una funzione i cui argomenti controllano la sua esecuzione e possono essere sovrascritti grazie ad una vulnerabilità di tipo buffer

overflow.
Nel primo attacco presentato è inoltre necessaria la presenza di una funzione che venga chiamata in più di un punto del programma, e che la funzione vulnerabile venga sfruttata per effettuare una jump tale da modificare il flusso di esecuzione del programma.
Gli attacchi presentati hanno la caratteristica comune di sfruttare le limitazioni o l'insufficiente livello di granularità dei modelli di IDS. Le informazioni o il tipo di algoritmo di modellazione utilizzato dell'IDS potrebbe infatti non segnalare come anomalie determinati tipi di attacco.
L'esempio in Figura 5.3 prevede un buffer overflow ed una jump in una funzione successiva saltando un controllo eseguito con un costrutto if. Tale attacco non viene rilevato oltre che dai modelli di IDS nemmeno dal VTPath, a meno che IP1 (riga 20) non venga innestata ad un differente livello di profondità, così che possa essere rilevata una anomalia nel contenuto dello stack.
Il modello implementato si ritiene abbia il massimo livello di granularità siccome è in grado di rilevare tutti gli attacchi che causano la deviazione del programma dal suo comportamento imparato. In molti casi gli IDS hanno un livello di granularità inadeguato che permette a certi attacchi di non essere rilevati.

5.1.4 Mimicry

Come spiegato nel capitolo 2 la forma più semplice di questo tipo di attacco chiamata traditional mimicry prevede la presenza di una vulnerabilità che possa permettere all'attaccante di modificare l'esecuzione del flusso del programma, in particolare l'attaccante eseguirà il codice dell'attacco mimando l'esecuzione delle syscall seguendo le tracce salvate dall'IDS per non generare un'anomalia.
In altre parole, vengono eseguite tutte e nello stesso ordine le syscall che sono state salvate dall'IDS, ma le syscall non necessarie all'attaccante per lanciare l'attacco vengono "nullificate", cioè vengono fatte fallire ad esempio passandogli i parametri scorretti.
Il modello VTPath, come spiegato nel capitolo 3, è in grado di rilevare anomalie avvenute nello stack e l'esecuzione di syscall diverse da quelle imparate nella fase di

```
1:  f(){
2:  ...
3:
4:  // read in some large string z, syscalls are fine here
5:  f0();
6:
7:  // important: f1() has no system calls
8:  // it copies z to x, z is larger than x, so x is overflowed;
9:  // after the ret instruction, the overflow code can
10: // jump anywhere within f(), as long as it is between
11: // f1() and the next system call;
12: // for example, the code can jump to IP1
13: f1();
14:
15: if( cond ){
16: // regular user privileges
17: ...
18: return;
19: }
20: IP1:
21: // superuser privileges
22: execve( "/bin/sh" );
23: }
```

Figura 5.3: Pseudo codice Attacco 3

learning. Sfruttando una vulnerabilità di tipo buffer overflow un attaccante per non generare un'anomalia deve innanzitutto non corrompere lo stack e fare in modo di eseguire la stessa system call che verrebbe eseguita nel normale flusso, eventualmente modificando solo i parametri. Una volta modificati i parametri, l'attaccante deve impostare un giusto[1] virtual stack per non creare un'anomalia. Questo virtual stack contiene il punto di ritorno del programma dopo l'esecuzione della syscall, il quale riporta il flusso del programma all'interno del codice originale, facendo perdere all'attaccante la possibilità di eseguire altro codice maligno.

Con un attacco di tipo traditional mimicry un attaccante è in grado di eseguire solamente una system call senza generare una anomalia al modello del VTPath. Come è facile capire, l'esecuzione di una sola system call non può essere sfruttata per lanciare

[1]Un virtual stack imparato nella fase di learning che segue il flusso del programma.

un attacco completo ed è comunque condizionata alla presenza di una vulnerabilità.
Kruegel et al. [12] hanno sviluppato uno strumento automatico che rende capace un attaccante di avere il pieno controllo del flusso di esecuzione dell'applicazione senza variare l'ordine delle syscall eseguite e la struttura dello stack, riguadagnando il flusso del codice dopo il controllo d'integrità dello stack effettuata dall'IDS. A questo recentissimo tipo di attacco sono vulnerabili tutti i tipi di IDS proposti in questa tesi, che rappresentano lo stato dell'arte degli HIDS.

5.1.5 Nullify VTPath

Analizzando il funzionamento del modello del VTPath e studiando alcuni tipi di attacchi, ci si è resi conto che di fronte a determinate strutture di un programma, tale modello non è in grado di rilevare la riesecuzione di una system call.
Si consideri il codice di un programma che permette di cambiare la precedente password di un utente con quella letta da rete, come riportato qui sotto.
Dalla funzione *main* viene richiamata la funzione *write_data* (riga 10) che esegue un ciclo while in cui esegue una sola syscall *write* per ogni ciclo. Tale funzione viene usata per scrivere il contenuto di un buffer in un file su disco.
Sempre nel main vengono eseguite due syscall *read*, che leggono lo username e la password da file e dalla rete rispettivamente. Se lo username e la password lette da rete corrispondono ad un utente del sistema (si controlla */etc/shadow*), viene aggiornata la password, cioè viene eseguita una syscall *write* (riga 17) e vengono aggiornate alcune strutture in memoria del sistema. In particolare viene eseguita una *strcpy* che permette ad un utente autorizzato di sfruttare una vulnerabilità di tipo buffer overflow per rieseguire la syscall *write* della riga 17 con i parametri modificati in modo da cambiare la password dell'utente root. L'esecuzione della funzione *write_data* genera i seguenti Virtual Stack List e Virtual Path dove W_n rappresenta il Virtual Stack List della system call write (riga 5) al ciclo n:

$$W_1 = \{w_0, w_1, w_2, w_3\}$$
$$W_2 = \{w_0, w_1, w_2, w_3\}$$

```
1: write_data(int vect[10])
2: {
3:          i=0;
4:          while(i<lenght(vect))
5:          write(sd,buf,1024);
6: }
7: main()
8: {
9:          ...
10:          write_data(vect);
11:          ...
12:          fd=open(/etc/shadow,w);
13:          read_socket(s_utente_password,new_password);
14:         read_file(fd,f_utente_password);
15:         if(s_utente_password=f_utente_password)
16:                  {
17:                         write(fd,utente,new_password);
18:                         strcpy(utente,last_user);
19:                 }
20: }
```

$W_3 = \{w_0, w_1, w_2, w_3\}$

...

(dove w_0, w_1 e w_2 sono gli indirizzi di ritorno di *main()*, *write_data()* e *write()* rispettivamente, e w_3 è il program counter corrente, cioè l'indirizzo della funzione di libreria condivisa che risulta essere lo stesso per tutte le invocazioni della stessa funzione.)
Il Virtual Path tra la prima e la seconda write risulta essere nullo siccome tutti gli indirizzi contenuti nei due Virtual Stack List risultano uguali.
Nel modello teorico presentato nell'articolo di Feng et al [7] non viene descritta la possibilità di avere un Virtual Path vuoto e quindi non è possibile dire come questo particolare caso venga gestito.
Assumiamo che venga salvato nel database il Virtual Path vuoto, come se fosse un Virtual Path normale.
Un attaccante potrebbe sfruttare tale Virtual Path vuoto e la particolare struttura di un programma, come quella presentata nell'esempio, per effettuare un attacco che non verrebbe rilevato dal modello.

Si ipotizzi che l'attaccante possieda un account sul sistema, quindi uno username e password validi. L'attaccante può sfruttare la vulnerabilità della strcpy alla riga 19 per effettuare un buffer overflow e modificare il valore del program counter in modo tale da ritornare ad eseguire la write di riga 17 con i parametri modificati in modo da cambiare la password dell'utente root.
L'ids in fase di detection all'esecuzione delle syscall read e write registrerebbe i seguenti Virtual Stack List:
(*read_file* rappresenta il Virtual Stack List della system call read alla riga 14, r_0 e r_1 sono gli indirizzi di ritorno di *main()* e *read()* rispettivamente e r_2 è il program counter corrente.)

$$read\_file=\{r_0, r_1, r_2\}$$

(*writefile* rappresenta il Virtual Stack List della system call write alla riga 17, wr_0 e wr_1 sono gli indirizzi di ritorno di *main()* e *write()* rispettivamente e wr_2 è il program counter corrente.)

$$write\_file=\{wr_0, wr_1, wr_2\}$$

Il Virtual Path *R-W* tra la syscall read e la write risulta essere:

$$R\text{-}W=r_2 \rightarrow Exit; r_1 \rightarrow wr_1; Entry \rightarrow wr_2$$

Come detto prima sfruttando il buffer overflow l'attaccante esegue una system call write modificando i parametri, e il detector calcolerebbe il Virtual Stack List:
(owr_0 e owr_1 sono gli indirizzi di ritorno di *main()* e *write()* rispettivamente e owr_2 è il program counter corrente.)

$$OVERFLOW\_write\_file=\{owr_0, owr_1, owr_2\}$$

che risulterebbe essere uguale a quello della system call write eseguita prima ed il Virtual Path calcolato tra le due risulterebbe nullo.
Il Virtual Path nullo era già presente nel database siccome era stato imparato tracciando la funzione *write_data*.
Visto che sia il Virtual Stack List che il Virtual Path generati dalla seconda system call

write risultano essere già presenti nel database, non viene generata nessuna anomalia e l'attacco non viene rilevato.
In pratica tutte le volte che viene eseguita una sola system call in un ciclo, il modello calcola un Virtual Path nullo per ogni coppia di stesse syscall, e registra questo Virtual Path nullo nel database.
In programmi con una certa struttura e la presenza di una vulnerabilità, sfruttando la quale si può cambiare il flusso di esecuzione del programma, si è in grado di ripetere l'esecuzione di una system call con parametri diversi senza generare anomalie.
Una soluzione per evitare questo tipo di attacco prevede che nella fase di learning si registri nel Virtual Stack List se una system call ha generato un Virtual Path nullo, il che significa che è stata eseguita almeno due volte in un ciclo in cui risultava essere l'unica system call.
Questa informazione permetterebbe di rilevare la ripetizione dell'esecuzione di una system call sfruttando il Virtual Path nullo.
L'attacco presentato e la relativa soluzione implementativa proposta sono stati testati con il prototipo implementato ed hanno confermato la debolezza del modello teorico all'attacco e l'efficacia della soluzione proposta.

5.2 Verifica Sperimentale

Il prototipo del modello di IDS implementato è stato provato su piccoli programmi e servizi scritti ad hoc per verificarne il comportamento e la corretta implementazione e una volta raggiunto un prototipo funzionante è stato testato facendogli tracciare le esecuzioni di una vera applicazione, un server web monoprocesso chiamato thttpd [4].
In una prima fase si sono tracciate solamente applicazioni monoprocesso siccome il modello teorico non fornisce informazioni su come gestire sia la fase di learning che quella di detection in caso di applicazioni multiprocesso. A completamento di tale modello teorico sono state fornite soluzioni implementative per tracciare servizi e applicazioni multiprocesso.
Thttpd è un server http con tutte le feature del caso, come supporto per il protocollo HTTP 1.1, CGI e così via. Questo webserver è stato scritto con un particolare riguardo

alle prestazioni, e le sue richieste in termini di memoria e CPU sono davvero minime. Gli esperimenti sono stati effettuati su di un Pentium 4 1900MHz, 512 MB di ram e con sistema operativo GNU/Linux Debian.
In un primo momento ci si è focalizzati sulla fase di learning, si è monitorato il webserver mentre pubblicava alcune pagine statiche e alcuni client le richiedevano, imparando e salvando nel database le tracce raccolte su più esecuzioni. In un secondo momento si è passati alla fase di detection cercando di capire se le anomalie segnalate erano dovute ad una non completa fase di learning piuttosto che ad errori di implementazione.
Da quanto si è potuto sperimentare se si effettua una buona fase di learning cercando di eseguire, per quanto possibile, tutti i possibili percorsi di esecuzione dell'applicazione, il modello risulta robusto e privo di falsi positivi.
Visto che la vecchia versione 2.21 di thttpd è nota per la presenza di vulnerabilità, sono stati provati alcuni attacchi di remote buffer overflow conosciuti, verificando che il modello rileva le anomalie avvenute nello stack, come previsto. Va precisato che i buffer overflow conosciuti non permettevano l'esecuzione di codice o l'esecuzione di una shell ma causavano solamente il crash del servizio.
Il server thttpd è stato anche modificato per inserire vulnerabilità ad hoc sfruttabili per effettuare attacchi di tipo IPE e Mimicry e per testare se realmente il modello fosse in grado di rilevare tali attacchi.
Per quanto riguarda gli attacchi di tipo IPE, è stata inserita una funzione vulnerabile prima di un costrutto *if()* e sono state modificate alcune funzioni che venivano richiamate. In pratica si è cercato di ricreare alcuni degli esempi di attacchi presentati prima. L'IDS è stato in grado di rilevare con precisione gli attacchi come previsto.
Per quanto riguarda gli attacchi di tipo traditional mimicry si è verificata la possibilità di modificare solo i parametri di una system call senza generare allarmi. Ulteriori modifiche venivano puntualmente rilevate causando anomalie.

5.3 Conclusioni

In questo lavoro di tesi è stato studiato ed analizzato un recente modello di Intrusion Detection System chiamato VTPath. Oltre alla comprensione e all'analisi delle

principali caratteristiche di funzionamento di questo modello sono stati analizzati altri modelli dello stato dell'arte degli Host Intrusion Detection System sottolineandone pregi e difetti. La fase di implementazione e sviluppo del modello hanno permesso una miglior comprensione dei meccanismi che regolano questo tipo di IDS. Sono state inoltre fornite soluzioni implementative non trattate nel modello teorico e tralasciate in letteratura ma che risultano determinanti nella valutazione dell'applicabilità del modello nei sistemi reali. Queste soluzioni hanno consentito di implementare un modello completo e funzionante su piattaforma GNU/Linux.
Nella parte di sperimentazione sono stati effettuati alcuni test per valutare la resistenza del modello di IDS ad attacchi noti o costruiti appositamente.
Analizzando il modello teorico è stato inoltre rilevato un particolare caso in cui è possibile rieseguire una stessa system call con parametri differenti, sfruttando un Virtual Path nullo. E' stata infine fornita una soluzione per evitare questo tipo di attacco.

Bibliografia

[1] *http://www.symantec.com*.

[2] *http://www.sqlite.org*.

[3] *http://www.cert.org/*.

[4] *http://www.acme.com/software/thttpd/*.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In USENIX *Security Symposium*, 2003.

[6] D. E. Denning, S. G. Akl, M. Heckman, T. F. Lunt, M. Morgenstern, P. G. Neumann, and R. R. Schell. Views for multilevel database security. *IEEE Trans. Software Eng.*, 13(2):129–140, 1987.

[7] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection using Call Stack Information. *IEEE Symposium on Security and Privacy, Oakland, California*, 2003.

[8] J. T. Giffin, S. Jha, and B. P. Miller. Detecting Manipulated Remote Call Streams. *11th USENIX Security Symposium*, 2002.

[9] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-Sensitive Intrusion Detection. *RAID Conference*, 2004.

[10] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[11] iSec.pl Development Team. kNoX - Implementation of non-executable Page Protection Mechanism. http://www.isec.pl/projects/knox/knox.html.

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, August 2005.

[13] E. A. O. Levy. Smashing the Stack for Fun and Profit. Phrack Magazine, Volume 0*x*07, Issue #49, Phile 14 of 16, 1998.

[14] B. Schenier. The process of security. *Information Security*, 2000.

[15] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. *IEEE Symposium on Security and Privacy, Oakland, California*, 2001.

[16] P. Szor. The art of computer virus research and defense. *Addison Wesley Professional*, 2005.

[17] T. L. K. . D. Team. The Linux Kernel 2.6. http://lwn.net/Articles/121845/, February 2005.

[18] T. O. D. Team. The OpenWall Project. http://www.openwall.com.

[19] T. P. Team. The PaX Project.

[20] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy, Oakland, California*, 2001.

[21] D. Wagner and P. Soto. Mimicry Attacks on Host Based Intrusion Detection Systems. *In Proc. Ninth ACM Conference on Computer and Communications Security.*, 2002.

[22] R. N. Wojtczuk. The Advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Magazine, Volume 0*x*0*b*, Issue 0*x*3*a*, Phile #0*x*04 of 0*x*0*e*, December 2001.

[23] H. Xu, W. Du, and S. J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. *RAID LNCS 3224 Springer-Verlag*, pages 21–38, 2004.