

Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA
Elettronica Applicata II

**LinSTAT: Un Sistema di Rilevamento degli
Attacchi Informatici**

TESI DI LAUREA IN INGEGNERIA INFORMATICA

Relatore: *Chiar. mo Prof.*
BRUNO RICCÒ
(Elettronica Applicata II)

Candidato:
MARCO COVA

Correlatore: *Chiar. mo Prof.*
OZALP BABAUGLU

Correlatore: *Chiar. mo Prof.*
GIOVANNI VIGNA

Sessione II

Anno Accademico 2002-2003

Indice

Elenco delle figure	v
Elenco delle tabelle	vii
1 Introduzione	3
1.1 Sistemi di Rilevamento delle Intrusioni	3
1.2 STAT	5
1.3 LinSTAT	7
2 An Introduction to Intrusion Detection	11
2.1 What is Intrusion Detection	11
2.2 Architecture	12
2.3 Taxonomy	14
2.3.1 Detection Method	14
2.3.2 Misuse-based Systems	14
2.3.3 Anomaly-based Systems	18
2.3.4 Behavior on Detection	19
2.3.5 Audit Source Location	19
2.3.6 Usage Frequency	22
2.4 Evaluation and Testing	22
2.5 Open Issues	27
2.5.1 System Effectiveness	27
2.5.2 Audit Throughput	27
2.5.3 IDS Cooperation and Alert Correlation	27
3 STAT	31
3.1 STAT Technique	31
3.2 STATL Language	33
3.2.1 Language Entities	33
3.2.2 Language Extension	37
3.2.3 Translation	37

3.2.4	Run Time Behavior	39
3.2.5	Example	40
3.3	STAT Core	43
3.4	Web of Sensors	44
3.4.1	MetaSTAT	45
3.4.2	Configuration of a STAT Sensor	46
3.5	Family of Sensors	48
3.5.1	USTAT	48
3.5.2	WinSTAT	49
3.5.3	NetSTAT	49
3.5.4	WebSTAT and logSTAT	49
3.5.5	AlertSTAT and afedSTAT	49
3.5.6	AODVSTAT and AgletSTAT	50
4	LinSTAT	51
4.1	Overview	51
4.2	Architecture	52
5	LinSTAT audit Data Source	55
5.1	LinSTAT Audit Source	55
5.1.1	What Type of Monitoring	56
5.1.2	What to Monitor	56
5.1.3	Where to Monitor	56
5.1.4	Implementation	57
5.2	Audit Data	58
5.2.1	Raw Audit Events	60
5.2.2	Audit Information	62
6	LinSTAT Extensions	73
6.1	LinSTAT Events	73
6.2	LinSTAT Factbase	76
6.3	LinSTAT Types and Predicates	79
6.4	The Tainting Mechanism	79
7	Testing	85
7.1	Performance Testing	85
7.1.1	Testing Methodology	85
7.1.2	Results	87
7.1.3	Components of System Overhead	87
7.2	Functional Testing	91
7.2.1	Scenario linux_buf_overflow	92

7.2.2	Scenario linux_anomal_execve	96
7.2.3	Scenario linux_system_program_write	100
7.2.4	Scenario restricted_read	103
7.2.5	Scenario traverse_vulnerability	106
7.2.6	Scenario qpopper	109
7.2.7	Scenario openssl	114
8	Conclusion and Future Work	119
	Bibliografia	121

Elenco delle figure

2.1	CIDF description of an IDS system.	13
2.2	Block diagram of a typical knowledge-based IDS.	15
2.3	Block diagram of a typical behavior-based IDS.	18
2.4	Classification of hits and misses in an IDS.	23
2.5	Sample ROC curves for three different IDSs.	25
3.1	Sample State Transition Diagram.	32
3.2	STATL translation process.	38
3.3	State Transition Diagram of <code>unix_ftp_write</code> scenario attack. . .	43
3.4	Configuration steps of a STAT based sensor.	47
3.5	Architecture of a web of sensors.	48
4.1	LinSTAT architecture	52
4.2	LinSTAT state machine diagram.	54
5.1	Cost of data transfer from kernel to user-space.	59
5.2	An example of an event entry in the audit trail.	60
6.1	UML diagram of <code>LINUX_Event</code>	75
6.2	A LinSTAT attack scenario that uses the tainting mechanism. . .	83
7.1	Job completion times in the Matmul test according to config- uration.	88
7.2	Result of Matmul test.	88
7.3	Job completion times in the Compile test according to config- uration.	89
7.4	Result of Compile test.	89
7.5	STD for the scenario <code>linux_buf_overflow</code>	92
7.6	STD for the scenario <code>linux_anomal_execve</code>	97
7.7	STD for the scenario <code>linux_system_program_write</code>	100
7.8	STD for the scenario <code>linux_restricted_read</code>	103
7.9	STD for the scenario <code>linux_traverse_vulnerability</code>	106

7.10	STD for the scenario linux_qpopper.	110
7.11	STD for the scenario linux_openssl.	115

Elenco delle tabelle

3.1	Dependencies and roles of STAT sensor modules.	44
4.1	Intrusion detection tools for Linux	52
4.2	Characteristics of LinSTAT	53
5.1	System calls audited by LinSTAT.	62
5.2	Audit data for I/O events.	63
5.3	Audit data for I/O events.	64
5.4	Audit data for CHANGE_OWNER events.	65
5.5	Audit data for DOUBLE_PATH_CLASS events.	67
5.6	Audit data for PROCESS_CONTROL_CLASS events.	68
5.7	Audit data for SUBSTITUTE_CLASS events.	68
5.8	Audit data for PROCESS_SPAWN_CLASS events.	69
5.9	Audit data for NET_CLASS events.	70
5.10	Audit data for ADMIN_CLASS events.	71
6.1	LinSTAT event types.	74
6.2	Correspondence between raw audit events and STAT events.	77
6.3	Predicates used to query the Factbase.	80
6.4	Predicates used to perform pattern matching on filenames.	80
6.5	Other LinSTAT predicates.	81
6.6	LinSTAT's tainting predicates.	82

Sommario

Questa tesi illustra la progettazione e l'implementazione di un Sistema di Rilevamento delle Intrusioni chiamato LinSTAT. LinSTAT è orientato al rilevamento di attacchi contro sistemi basati su Linux.

LinSTAT è basato su STAT (State Transition Analysis Technique), un framework per la realizzazione di sistemi di rilevamento di intrusioni sviluppato presso l'Università della California di Santa Barbara. Lo STAT Framework implementa una serie di componenti indipendenti dal particolare sistema oggetto di monitoring ed è stato utilizzato per monitorare sistemi tra loro molto diversi, dal semplice applicativo, per esempio un server web, al singolo computer, ad una rete costituita di svariati computer.

La realizzazione di LinSTAT ha richiesto lo sviluppo di un meccanismo per l'auditing di sistemi Linux a livello di kernel e di un sensore, basato su STAT, in grado di analizzare le informazioni di audit generate. LinSTAT è stato sottoposto a test di performance e funzionali. Svariati scenari che modellano vari attacchi contro un tipico sistema Linux sono stati creati, quando possibile, sulla base di exploit disponibili su Internet.

Capitolo 1

Introduzione

In questo Capitolo si delineano brevemente i punti fondamentali sviluppati in questa tesi. Una trattazione più approfondita di tutti gli argomenti affrontati è condotta, in Inglese, a partire dal Capitolo successivo.

1.1 Sistemi di Rilevamento delle Intrusioni

Negli ultimi anni, il numero di attacchi ai danni dei sistemi informatici è andato rapidamente aumentando. Questi attacchi sfruttano un ampio spettro di vulnerabilità, per esempio, errori di programmazione, uso di protocolli di comunicazione insicuri, o un'errata applicazione di primitive crittografiche. Indipendentemente dalle specifiche vulnerabilità sfruttate negli attacchi, il loro numero e distribuzione lasciano pensare che la progettazione e implementazione di sistemi assolutamente sicuri siano obiettivi difficilmente raggiungibili nel prossimo futuro.

È chiaro che a fronte di sistemi di prevenzione tutt'altro che perfetti, si rende sempre più necessaria la disponibilità e l'utilizzo di sistemi in grado di rilevare eventuali attacchi e fornire adeguate contromisure. Un Intrusion Detection System (IDS) è un sistema che permette di monitorare le attività di un sistema informatico, rilevare attacchi diretti contro di esso e mettere in atto azioni correttive.

Negli ultimi vent'anni gli Intrusion Detection System sono stati oggetto di numerose ricerche che hanno portato ad un significativo sviluppo di questo settore della sicurezza informatica. In particolare, sono stati individuati diversi approcci al problema di rilevare intrusioni e attuare contromisure, si sono studiate architetture software per sistemi IDS, e sono stati proposti alcuni metodi per permettere l'interazione e la cooperazione tra sistemi diversi.

Di seguito si discutono brevemente questi aspetti introduttivi e generali di questo campo di ricerca.

Rilevamento delle intrusioni

Per quanto riguarda il rilevamento delle intrusioni, si sono affermati due approcci complementari: *misuse detection* e *anomaly detection*. Sistemi che impiegano l'approccio di misuse detection cercano essenzialmente di definire con esattezza tutte le azioni che rappresentano un uso potenzialmente nocivo del sistema. In pratica, in questi sistemi, tutti gli attacchi noti sono modellati e memorizzati in una knowledge base e l'attività del sistema è messa a confronto con tali descrizioni. Se il confronto è positivo, l'attività è considerata istanza di un attacco e, conseguentemente, sono messe in atto le contromisure previste. Sistemi basati su questo approccio ottengono tipicamente buoni risultati in termini di accuratezza, ovvero vengono generati un numero limitato di falsi allarmi, ma attacchi nuovi potrebbero non essere rilevati.

Sistemi basati sull'approccio di anomaly detection modellano, invece, il comportamento atteso di utenti e applicazioni, e interpretano deviazioni da tale comportamento "normale" come manifestazioni di un attacco. Il vantaggio principale di questo approccio consiste nella possibilità di poter rilevare attacchi nuovi. Tuttavia, questi sistemi generano, tipicamente, molti falsi allarmi e, di conseguenza, la loro accuratezza non è particolarmente elevata.

Contromisure

Una volta che un'intrusione è stata rilevata, si pone il problema di quali contromisure mettere in atto. A questo proposito, si è soliti distinguere tra IDS *attivi* e *passivi*. Gli IDS passivi si limitano a generare un alert e ad inviarlo al Security Officer, che dovrà ispezionarlo manualmente e decidere l'azione appropriata da intraprendere. È evidente che in un sistema di questo tipo, tra il rilevamento di un attacco e la messa in atto di una contromisura può intercorrere un intervallo di tempo significativo. I sistemi attivi, al contrario, sono in grado non solo di rilevare un attacco e allertare il Security Officer, ma possono anche attivare una serie di misure difensive, quali la modifica di permessi sui file, l'attivazione di nuove regole del firewall, l'uccisione di processi, l'abbattimento di connessioni di rete.

Architettura

I sistemi IDS rappresentano una sfida interessante anche dal punto di vista dell'ingegneria del software. Sono spesso, infatti, sistemi complessi, molto

vasti, costituiti di numerosi componenti e con precisi requisiti di performance. Ci si sofferma qui brevemente sugli aspetti architetturali.

La maggior parte dei sistemi IDS è realizzata utilizzando uno schema architetturale comune ed indipendente dalle caratteristiche specifiche del singolo sistema, quale il metodo di rilevamento o l'ambito di utilizzo. Tale schema prevede la presenza di quattro componenti fondamentali: un generatore di eventi, che produce dati di monitoraggio (audit); un analizzatore di eventi, che analizza i dati di monitoraggio e determina se questi contengono manifestazioni di attacchi; un modulo delle risposte, incaricato di mettere in atto contromisure ad un attacco rilevato; ed infine, un modulo per la memorizzazione dei dati, utilizzato con fini di archivio e analisi post-mortem.

Integrazione di sistemi diversi

Un aspetto che sta assumendo sempre maggior importanza è l'integrazione di IDS diversi. L'uso di IDS diversi all'interno di una singola organizzazione ha molteplici ragioni: la necessità di monitorare una rete di computer eterogenea e in cui coesistono diverse risorse da difendere può richiedere l'uso di IDS specializzati nel monitoraggio e nella difesa di un particolare sottosistema; la volontà di sfruttare sistemi con punti di forza diversi, ad esempio IDS basati su tecniche di rilevazione misuse e anomaly detection; il desiderio di ottenere descrizioni degli attacchi di più alto livello.

Il requisito minimo necessario per permettere la cooperazione di IDS diversi è l'adozione di un formato degli alert comune che permetta ad un analizzatore di correlare l'informazione presente negli alert generati dai vari sistemi.

Il Capitolo 2 presenta una più dettagliata introduzione ai Sistemi di Rilevamento delle Intrusioni e alle loro caratteristiche generali.

1.2 STAT

In questa Sezione si descrive brevemente lo STAT Framework. Un'analisi più dettagliata è presentata nel Capitolo 3.

Motivazioni e obiettivi

Lo sviluppo di sistemi IDS deve far fronte a due problematiche tra loro ortogonali. Da una parte, è necessario progettare e realizzare degli IDS specializzati per la difesa di sistemi e piattaforme specifiche e molto diverse tra loro, ad esempio, un programma come un web server e una rete di computer, o,

ancora, un computer basato su Windows e uno basato su Linux. D'altra parte, riprogettare e re-implementare da capo un sistema complesso, come può essere un IDS, ogni volta che cambia il target applicativo sarebbe, evidentemente, un processo lento e costoso. Ecco, quindi, giustificata la necessità di disporre di librerie e framework che realizzino i componenti indipendenti dalla specifica piattaforma o ambiente di elaborazione e che siano, quindi, riusabili in progetti diversi.

Componenti del framework

STAT è un framework per lo sviluppo di sistemi IDS, progettato e realizzato presso l'Università della California a Santa Barbara. STAT fornisce un runtime generico, indipendente, cioè, dal particolare dominio applicativo, che può essere esteso per implementare sistemi adatti ad uno specifico ambiente operativo.

Lo STAT framework è basato sui seguenti concetti: la tecnica STAT, un metodo per modellare attacchi informatici; il linguaggio STATL, un linguaggio per rappresentare attacchi secondo la tecnica STAT; lo STAT Core, che rappresenta il runtime del linguaggio STATL; infine, MetaSTAT, un'infrastruttura per il collegamento e la comunicazione tra sensori basati su STAT.

La tecnica STAT è un metodo per descrivere in modo astratto intrusioni informatiche. Un attacco è modellato come una successione di stati e transizioni di stati. Gli stati rappresentano la condizione di sicurezza di un sistema ad un certo istante temporale. Le transizioni modellano l'evoluzione del sistema da uno stato ad un altro. Una transizione avviene in corrispondenza di specifici eventi, quali la lettura di un file, l'apertura di una connessione di rete, ecc. La modellazione di un attacco è detta scenario d'attacco.

STATL è un linguaggio estensibile che è usato per rappresentare attacchi modellati secondo la tecnica STAT. Il linguaggio definisce le caratteristiche indipendenti dal dominio applicativo previste da STAT, ad esempio, stati, transizioni, timer, ecc. Il linguaggio deve essere esteso per esprimere caratteristiche che sono specifiche di un determinato dominio. In particolar modo, le estensioni permettono di definire nuovi eventi, tipi e predicati.

Lo STAT Core rappresenta il runtime del linguaggio STATL e ne implementa le caratteristiche indipendenti dal dominio applicativo. Inoltre, il Core effettua il rilevamento delle intrusioni vero e proprio, mettendo a confronto uno stream di eventi con gli scenari d'attacco definiti.

MetaSTAT, infine, è un'infrastruttura per il supporto di IDS distribuiti. I componenti di cui è costituita cooperano al fine di permettere la gestione e la riconfigurazione dinamica dei sensori che costituiscono il sistema distribuito.

Lo STAT Framework è discusso in maggior dettaglio nel Capitolo 3.

1.3 LinSTAT

Come visto nella Sezione precedente, lo STAT Framework fornisce una serie di costrutti e tecniche per modellare e implementare la parte indipendente dal dominio di un IDS. Tuttavia, il rilevamento delle intrusioni è condotto in domini applicativi specifici. LinSTAT estende STAT in modo tale da realizzare un sistema di rilevamento delle intrusioni specializzato per sistemi basati sul sistema operativo Linux. Discuteremo di seguito il generatore di eventi utilizzato da LinSTAT e le estensioni a STATL.

Dati di audit

Per quanto riguarda la generazione di dati di audit, si è deciso di monitorare il sistema a livello di invocazioni di system call. Quando viene invocata una system call, LinSTAT raccoglie delle informazioni riguardanti il processo invocante, l'utente responsabile del processo, gli oggetti, ad esempio, file, socket, ecc., coinvolti nella chiamata di sistema, il valore di ritorno, e altri dati che possono essere ottenuti direttamente dai parametri passati alla system call. L'informazione raccolta viene incapsulata in un evento e propagata agli altri componenti del sistema di rilevamento delle intrusioni, in particolare, verso il modulo analizzatore.

Questo approccio è basato sull'assunzione che il sistema si trova inizialmente in uno stato di non compromissione e ogni processo che intende modificare lo stato del sistema, in particolare, per condurlo in uno stato di compromissione, lo può fare solamente attraverso invocazioni di system call.

Per ragioni di efficienza, disponibilità delle informazioni volute, e resistenza ad attacchi, il monitoraggio delle system call è effettuato in kernel space. Nell'implementazione corrente, un modulo per il kernel di Linux intercetta le system call, raccoglie l'informazione desiderata associata con ciascuna di esse, e rende disponibili in user space, attraverso il filesystem virtuale */proc*, i dati ottenuti. Il modulo monitora system call che permettono all'utente di interagire con il filesystem (`open`, `creat`, `mkdir`, `unlink`, `mknod`, `rmdir`, `chown`, `chmod`, `symlink`, `link`, `rename`, `truncate`, `mount`, `umount`), eseguire o terminare processi (`execve`, `exit`), generare nuovi processi (`fork`, `vfork` e `clone`), cambiare i privilegi di un processo (`setuid` e simili), accettare o stabilire connessioni di rete (`connect` e `accept`), e altre ancora (`create_module`, `delete_module`, `reboot`).

Per maggiori dettagli sull'implementazione del modulo di generazione dei

dati di audit, sui dati raccolti in corrispondenza di ogni system call, e per un esempio di evento, si rimanda al Capitolo 5.

Estensioni a STAT

LinSTAT estende STAT con un provider di eventi e con la definizione di nuovi eventi, tipi e predicati.

Il provider di eventi raccoglie i dati forniti dal modulo di intercettazione delle system call descritto sopra, incapsula i dati di audit in un evento STAT e lo propaga allo STAT Core. Un evento STAT modella un evento in un formato indipendente dal particolare dominio applicativo e analizzabile dallo STAT Core.

Gli eventi definiti da LinSTAT modellano l'invocazione delle system call elencate sopra. Sono, così, introdotti gli eventi di lettura e scrittura di un file, di creazione e terminazione di un processo, ecc. I predicati di LinSTAT consentono di testare varie condizioni su un evento e di manipolarne campi specifici, contenenti informazioni, ad esempio, su nomi di file e nomi utente.

LinSTAT introduce, inoltre, un meccanismo, detto di *tainting*, che consente di monitorare e tenere traccia dei processi creati su iniziativa di un utente remoto e di associarli all'indirizzo IP del sito da cui tale utente è connesso. Un'applicazione di questo meccanismo è la definizione e l'applicazione di policy di sicurezza differenti per utenti remoti e utenti locali. Questo consente, ad esempio, di stabilire che l'accesso ad alcune risorse del sistema è consentito ad utenti locali, ma non a quanti vi si collegano remotamente.

Tutti i dettagli relativi alle estensioni di LinSTAT allo STAT Framework e al meccanismo di tainting si possono trovare nel Capitolo 6.

Testing

LinSTAT è stato sottoposto a test volti a valutarne l'impatto sulle prestazioni del sistema su cui è in funzione. I test hanno confermato che l'overhead causato da LinSTAT è funzione del tipo di applicazioni in uso sul sistema, e, in particolare, è tanto maggiore quante più sono le invocazioni di system call effettuate dai processi in esecuzione. Tipicamente, questo si traduce in un maggior impatto per processi di tipo I/O bound e minore per processi CPU bound. L'overhead, inoltre, è risultato crescente al crescere del numero di scenari caricati e, quindi, del numero di attacchi che il sistema è in grado di rilevare. Nei test effettuati, a circa 15 scenari caricati corrisponde, nel caso peggiore, un overhead del 5-6%.

Il Capitolo 4 presenta un'introduzione più dettagliata di LinSTAT e della

sua architettura. Nel Capitolo 5 si approfondisce l'esame della sorgente di eventi di audit. Il Capitolo 6 completa l'analisi dei componenti necessari alla integrazione di LinSTAT nello STAT Framework. Infine, tutte le informazioni relative al testing di LinSTAT e agli scenari d'attacco sviluppati si possono trovare nel Capitolo 7.

Capitolo 2

An Introduction to Intrusion Detection

Intrusion Detection Systems (IDSs) have gained more and more relevance in the last decade. After the first articles proposing the very idea of intrusion detection in the '80s [5, 19], the field has rapidly evolved and today it is explored by many research projects and commercial products [6, 3].

In this chapter we present an overview of IDSs. We analyze the motivations behind their use, present one possible taxonomy of IDSs, and discuss evaluation and testing methodologies of system implementation. We conclude indicating open issues in the IDS field.

2.1 What is Intrusion Detection

Intrusion Detection Systems are software applications dedicated to detect intrusions against a target computer network. They analyze data provided by monitoring systems and try to identify manifestations of malicious activity. When this happens, they raise an alarm and may take actions in order to block the alleged attack.

Intrusion detection is an approach that is complementary with respect to mainstream approaches to security, such as authentication and access control. It is based on the consideration that prevention alone is not enough to secure computer systems:

1. Surveys have shown that most computer systems are flawed by vulnerabilities, regardless of manufacturer or purpose [47], that the number of security incidents is continuously increasing [12], and that users and administrators are generally very slow to apply fixes to vulnerable sys-

tems [69]. As a consequence, many observers believe that computer systems will never be absolutely secure [10].

2. Deployed security mechanism, e.g., authentication and access control, may be disabled as a consequence of misconfiguration or malicious actions.
3. Users of the system may abuse their privileges and perform damaging activities.
4. Even if an attack is not successful, in most cases it is useful to be aware of the compromise attempt.

Intrusion Detection Systems have been designed to address the issues described above. As such, they are not intended to replace traditional security methods, but rather to complete them.

2.2 Architecture

There exist many IDSs, based on different conceptual frameworks. It is still possible, however, to recognize a common architecture that underlies all intrusion detection systems. We will present the main components of IDSs and their functionality.

We will use the terminology introduced by the working group on the Common Intrusion Detection Framework (CIDF) [63]. CIDF models an IDS as an aggregate of four components with specific roles:

- *Event boxes (E-boxes)*. The role of Event boxes is to generate events, by processing raw audit data produced by the computational environment. A common example of an E-boxes is a program that filters audit data generated at the C2 level of TCSEC [61]. Another example is a network sniffer that generates events based on the network traffic.
- *Analysis boxes (A-boxes)*. The role of an Analysis box is to analyze the events provided by other components. The results of the analysis are sent back in the system as other events, typically representing alarms. Usually A-boxes analyze simple events supplied by E-boxes. Some A-boxes analyze events produced by other A-boxes and operate at a higher level of abstraction.
- *Database boxes (D-boxes)*. Database boxes simply store events, guaranteeing persistence and allowing post-mortem analysis.

- *Response boxes (R-boxes)*. Response boxes consume messages that carry directives about actions to be performed as a reaction to a detected intrusion. Typical actions include killing processes, resetting network connections, and modifying firewall settings.

Figure 2.1 presents an IDS system where two E-boxes monitor the environment and deliver audit events to two A-boxes. These A-boxes analyze the audit data and provide their conclusions to a third A-box that correlates the alerts, stores them in a D-box and controls a R-box. Dashed lines are used to indicate exchange of events, solid lines represent the exchanging of raw audit data.

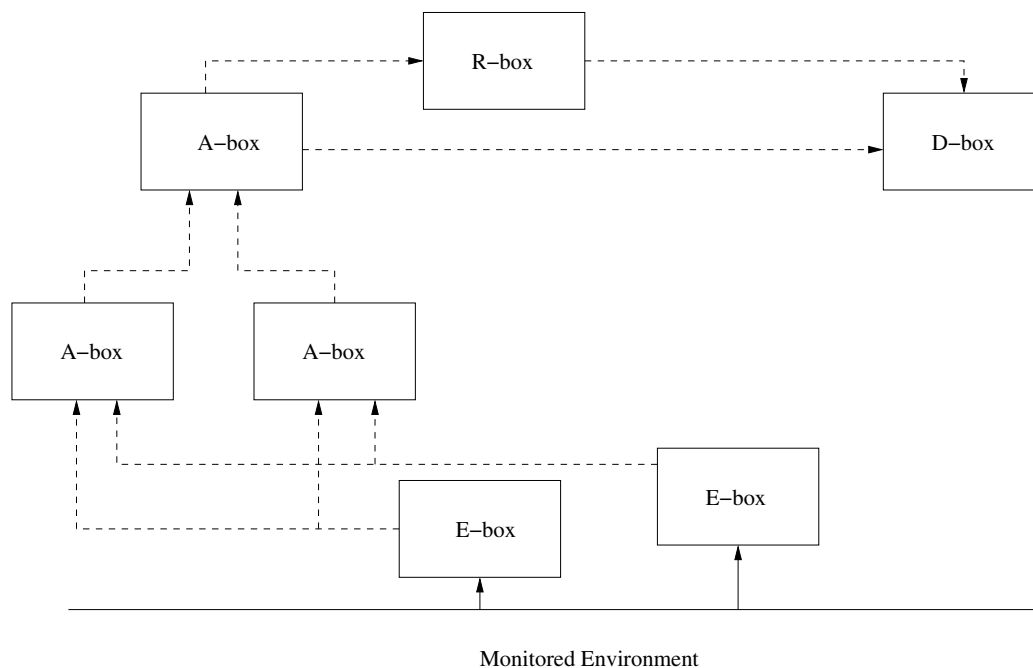


Figure 2.1: CIDF description of an IDS system.

Note that components are logical entities which produce or consume events. The CIDF model doesn't mandate what their implementation should be. It only states their roles and interactions. They can be realized as a single process on a single computer or as a collection of cooperating processes spanning multiple computers.

2.3 Taxonomy

Intrusion Detection Systems may be classified according to different characteristics [55, 3, 8]. The following ones seem to be particularly characterizing [16]:

- *Detection method.* It defines the philosophy on which the A-box is built. Two approaches have been proposed: when the IDS defines what is “normal” in the environment and flags as attacks deviations from normality, it is qualified as *anomaly-based*. When the IDS explicitly defines what is “abnormal”, using specific knowledge about the attacks in order to detect them, it is called *misuse-based*.
- *Behavior on detection.* It defines a characteristic of the R-box. It is said to be passive if the system just issues an alert when an attack is detected. If more proactive actions are taken, e.g., disconnecting users, shutting network connections, it is said to be active.
- *Audit source location.* It specifies where the E-box takes audit data from. We distinguish between host-based IDSs, which deal with audit data generated on a single host, e.g., C2 audit trail; application-based IDSs, which work on audit records produced by a specific application; and network-based IDSs, which monitor network traffic.
- *Usage frequency.* It discriminates between systems that analyze the data in real-time and those that are run periodically (off-line). It specifies how often the A-box analyzes data collected by other parts of the system.

We will analyze these characteristics more in detail in the following sections.

2.3.1 Detection Method

Detection can be performed according to two complementary strategies: (1) defining what is the manifestation of an attack and searching for an occurrence of the attack or (2) defining what is the normal behavior on the system and searching for activities that deviate from it.

2.3.2 Misuse-based Systems

Misuse-based systems are equipped with a database of information (the knowledge base) that contains a number of attack models (sometimes called

“signatures”). The audit data collected by the IDS is compared with the content of the database and, if a match is found, an alert is generated. Events that don’t match any of the attack models are considered part of legitimate activities.

The main advantage of misuse-based systems is that they usually produce very few false positives: attack description languages usually allows to model attacks at such a fine level of detail that only a few legitimate activities match an entry in the knowledge base.

However, this approach has drawbacks as well. First of all, populating the knowledge base is a difficult task. Furthermore, misuse-based systems cannot detect previously unknown attacks, or, at most, they can detect only new variations of previously modeled attacks. Therefore, it is essential to keep the knowledge base up-to-date when new vulnerabilities and attack techniques are discovered.

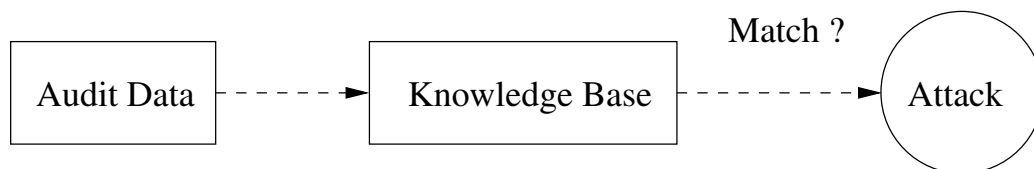


Figure 2.2: Block diagram of a typical knowledge-based IDS.

A number of different techniques have been proposed for performing misuse-based intrusion detection [37, 50, 45, 71, 60]. Before briefly discussing the most common techniques, it is worth discussing how they can be differentiated on the basis of state maintenance and the importance of this characteristic on the overall intrusion detection system.

Stateless Techniques

A *stateless* intrusion detection system treats each event independently of the others. When the processing of the current event is completed, every information regarding this event is discarded.

This simplifies the design of the system, specifically of the A-box, because it doesn’t need to allocate and maintain memory to keep information about past activity. Stateless systems usually have very good performance in terms of speed of processing because the analysis step is reduced to simple lookups in the knowledge-base and matches against the current event, with no additional operations needed.

However stateless systems have important limitations. In the first place, they are not complete, in the sense that there exist classes of attacks that they

cannot detect. For example, multi-step attacks are malicious activities that require a series of steps to be completed. Actions involved in each step are not per se intrusive and become malicious only when executed in the correct and complete sequence. Because the system has no memory of past events, it lacks the possibility to keep track of steps. Note it would be possible to model the entire attack on the basis of the last step, thus considering the action involved in this step as malicious, and inserting it in the attack knowledge-base. However, because this action alone is not intrusive, it is likely to be performed also as part of legitimate activities. Thus, this approach would possibly generate an unacceptable number of false positives, thwarting the main advantage of misuse-based intrusion detection.

Furthermore, stateless systems are subject to a class of attacks whose aim is to trigger the generation of a flood of alerts. A number of tools have been proposed that analyze the database of attacks and automatically generate events or series of events that conform to the attack descriptions. The event stream so synthesized forces intrusion detection systems to generate a large number of detection alerts. The resulting “alert storm” has been used to desensitize intrusion detection system administrators and hide attacks in the event stream [62, 59, 74, 77].

Stateful Techniques

Stateful intrusion detection systems maintain information about past events. As a consequence, the effect of a certain event on the system is not independent of its position in the event stream. While this approach adds an additional level of complexity to the design and implementation of A-boxes, it provides significant advantages. In particular, stateful tools are able to model and detect attacks that involve many steps. Furthermore, they are less prone to the alert storm attacks described in the previous paragraph, because it is more difficult to develop a program that is able to exactly reproduce the single steps of a modeled attack.

Signature-based Systems

In signature-based systems, abstractions (“signatures”) of attacks are stored in the knowledge base¹. Signatures are used to summarize in a compact format all the information necessary to describe attacks. For example, a network intrusion detection system may store in the knowledge base the content of network packets involved in known attacks. Signatures are stored

¹Note that this technique shares many ideas with the method of signature files in Information Retrieval. For a discussion of signature files in IR, refer to [24].

in a format that allows straightforward comparison with information found in the audit trail. The audit trail is checked against entries in the signature file: if a match is found, the system raises an alarm. As described here, signature-based systems implements stateless intrusion detection.

Signature based systems have gained popularity because they are easy to develop, give accurate feedback on alarms and are usually spare of computational resources.

They have disadvantages too:

- The description of an attack is usually a very low level description and thus, possibly, difficult to determine or interpret.
- Every attack or variation of an attack requires a new signature to be added to the knowledge base, thus its size can become unmanageable.
- The more specific a signature the less false positives are generated. But also, the more specific a signature the easier for an attacker to create a slightly different version of an attack that doesn't match the signature and thus goes unnoticed. Such an attack is said to be polymorphic.

Example of signature-based systems include Snort [71], EMERALD [60] and many commercial products.

State Transition Systems

State transition methods model an activity on the system as a series of state transitions, where a *state* is a snapshot of the system representing the value of all the memory locations on the system. Malicious activities move the state of the system from an initial safe state to a final compromised state, possibly passing through a number of intermediate states. This technique requires the analyst to identify those transitions that are critical to lead the system in a compromised state. The IDS will then search for such transitions. Of course, state transition systems are stateful tools.

This approach is appealing because it allows high-level, even graphical, description of attacks. It has very high expressive power, meaning that arbitrarily complicated attacks can be modeled and detected. In particular, multi-steps attacks fit very well the state transition modeling technique. It also provides very detailed feedback on the generated alerts, because the entire list of actions that caused the alarm to be triggered can be easily provided. Furthermore, it allows to deploy a response before an attack reaches its final step, thus allowing to effectively prevent the attack rather than only detecting it.

The main disadvantage of state transition technique is that its computational requirements can be high if the system has to keep track of many states.

2.3.3 Anomaly-based Systems

Anomaly-based systems are based on the assumption that all anomalous activities are malicious². Thus, they first build a model of the normal behavior of the system and then look for anomalous activities, i.e., activities that do not conform to the established model. Anything that doesn't correspond to the system profile is flagged as intrusive [32]. Many systems have been built following this approach, e.g., [33, 27, 80, 43].

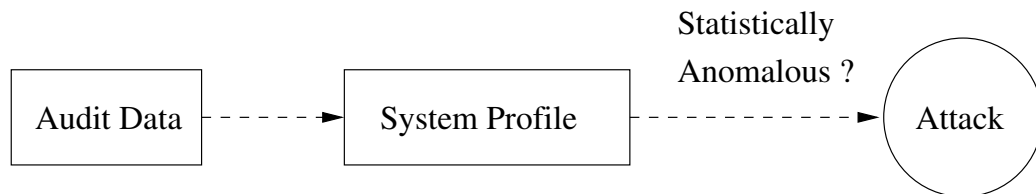


Figure 2.3: Block diagram of a typical behavior-based IDS.

The systems based on this approach may be able to detect previously unknown attacks. On the other hand, anomaly-based systems are prone to the generation of many false positives, that is their accuracy is typically low.

These systems may also suffer from specific problems:

- The amount of time required to teach to the system the normal behavior might be long.
- The behavior of the monitored environment might change during a period of time, so the system has to be re-trained.
- If the training set contains attacks, the system will consider malicious behavior normal.

²Note that “All malicious activities are anomalous” is different from “All anomalous activities are malicious”. The former implies that do not exist malicious activities that are not anomalous and thus assumes that systems that detect every anomaly also detect every attack. On the other hand, the latter leaves open the possibility that there exist attacks that do not appear as anomalous activity. Thus, the “all malicious is anomalous” assumption postulates that anomaly-based systems can detect every possible attacks. The “all anomalous is malicious” asserts that attacks might go undetected by an anomaly-based IDS.

- It may be possible to perform an attack within the boundaries of “normality”.

Statistical analysis has shown that, under reasonable assumptions, the probability $P(\text{Intrusion}|\text{Alarm})$, i.e., the probability that an alarm really indicates an intrusion, is dominated by the false alarm rate rather than the true positive detection rate [7]. This is a consequence of the *base-rate fallacy*: typical traffic shows a huge number of benign activities and only a few malicious events. The problem is that most systems today have poor performance in terms of suppression of false alarms.

A number of studies have indicated that most network systems are intrinsically characterized by a high rate of change and diversity [26]. Moreover, ambiguities in underlying protocols and differences in application programs exist [9]. Thus, detecting anomalous behavior in certain environments is very difficult, because anomalies are characteristic of the environment itself.

Finally, some new attack strategies have emerged that mimic legitimate activities and thus go undetected [86, 79]. While these studies were applied to a specific anomaly-based IDS, they are based on general concepts that seem to be applicable to the whole class of anomaly-based IDSs.

2.3.4 Behavior on Detection

Most intrusion detection tools are passive: when an attack is detected they generate an alert without further trying to oppose the attack. This requires the Security Officer to manually inspect the alert and take appropriate action. This can introduce a significant delay in the process of dealing with the attack.

A number of IDSs have proactive capabilities, e.g., they can change the security posture of a protected network to react to the detected attack. For example, they may modify file permissions, add firewall rules, kill processes or shutdown network connections. It is important to note, however, that automatic countermeasures could, in certain cases, be leveraged by the attacker to damage the system itself or to cause a denial of service.

2.3.5 Audit Source Location

Intrusion Detection Systems can be characterized according to the source of the events they analyze. Typical system classes include host-based IDSs, network-based IDSs, application-based IDSs, and correlation systems. These are discussed in detail in the next paragraphs.

Host-based IDSs

Host-based IDSs (HIDS) detect attacks against a specific host analyzing audit data produced by the host operating systems. Audit sources include:

- System information. Operating systems make available to user-space information about their internal working and security-relevant events. There exist programs that collect and show this information, e.g., *ps*, *vmstat*, *top*, *netstat*. The information provided is usually very complete and reliable because it is retrieved directly from the kernel. Unfortunately, few operating systems provide mechanism to systematically and continuously collect this information.
- Syslog facility [54]. Syslog is an audit facility provided by many UNIX-like operating systems. It allows programmers to specify a text message describing an event to be logged. Additional information, like the time when the event happened and the host where the program is running, is automatically added. Because of its simplicity, it is used extensively by applications, so it may provide much information. However, applications usually log information valuable for debugging purposes that is not necessarily tailored to the needs of intrusion detection. Furthermore, a specific audit format is not imposed by the facility but changes according to the program that uses it, so that it may be difficult to extract audit data from logs and perform sophisticated analysis. Finally, the logged information can be easily polluted by messages crafted by an attacker to cover his tracks.
- C2 audit trail. Some operating systems comply with the C2 level of TCSEC and thus monitor the execution of system calls. The data obtained is accurate because it comes directly from within the kernel.

Application-based IDS

Application-based IDSs detect attacks against a specific application. The audit information necessary to perform intrusion detection is usually obtained either using the already described syslog facility or instrumenting the application with specific audit mechanisms. The following discussion will focus on the latter approach.

Adding audit mechanisms to an existing application requires to modify the application so that it produces audit information in correspondence of security-relevant events. This can be accomplished in different ways:

- Directly modifying the application source code to include auditing code. This approach requires that the application code be available and modifiable.
- Interposing code responsible to extract audit information in correspondence of interfaces used by the application, e.g., the system call or the standard C library interface. One disadvantage of this approach is that applications other than the one to be monitored could be affected, e.g., in terms of performance loss, by the use of the interposition mechanism.
- Using extension hooks provided by the application itself to implement the audit collection functionality. This approach guarantees great flexibility but not all the applications offer extension hooks.

A description of a tool performing intrusion detection at the application level can be found in [4]. This system leverages the extension mechanism of the Apache web server to implement an audit data source. The collected audit information is used to monitor the behavior of the web server.

Network-based IDSs

Network-based IDSs (NIDSs) detect attacks by analyzing the network traffic exchanged on a network link. Therefore, in network-based IDSs, the E-boxes are simply network sniffers.

The analysis of the content of network packets can be performed at different levels of sophistication, e.g., performing simple pattern matching on the header and/or the payload of a packet or exploiting knowledge about the protocol followed by the communication. Higher-level analysis supports more sophisticated analysis of the data, but it is usually slower and requires more resources.

Network-based IDSs are very appealing because they are easy to deploy, and have a minimal or no impact the on monitored hosts. On the other hands, changes in network technology could impair the usefulness of NIDSs:

- High-speed networks might represent an issue, increasing the network throughput beyond the capabilities of sniffers.
- Switched networks make more difficult to choose the location where the NIDSs should be placed.
- The adoption of encryption of the communications reduces or completely prevents NIDSs from accessing the content of network connections.

Furthermore, studies have shown that, if particular care is not taken, NIDSs are vulnerable to a class of attacks, called insertion and evasion, which take advantage of the physical and logical separation of the NIDSs from their monitored hosts to undermine their detection capability. [67, 56, 30].

Correlation Systems

Correlation systems analyze alerts generated by other IDSs. Correlation systems perform the following functions:

- Alert clustering. Alerts that refer to the same occurrence of an attack are clustered together.
- Alert merging. A new alert is generated carrying the information contained in various alerts belonging to the same cluster.
- Alert correlation. Alerts that refer to different steps of the same occurrence of an attack are grouped in order to clarify the intentions of the attacker.

A more detailed discussion on alert correlation is postponed to Section 2.5.

2.3.6 Usage Frequency

Dynamic IDS tools analyze in real-time the activity of the system to be protected. Audit data is examined as soon as it is produced. The advantages of this approach are that the system activities can be analyzed timely and, thus, a proper response can be issued when an attack is detected. On the other hand, real-time collection and analysis of audit data may have a significant overhead.

Static tools are run off-line at specific intervals. They analyze a snapshot of the system state and produce an evaluation of the security of that state. They don't provide any security in between two consecutive runs and, thus, in case of a successful attack, they can be used only for post-mortem analysis. However, running only occasionally, they may perform a more thorough analysis without having an unacceptable impact on the performance of the monitored system.

2.4 Evaluation and Testing

Before discussing the issues of evaluation and testing of IDSs, it is necessary to introduce some terminology.

We define *false positives* (F_P) legitimate activities incorrectly flagged as malicious and *false negatives* (F_N) attacks incorrectly flagged as legitimate, that is, attacks that are not detected at all. Analogously, *true negatives* are legitimate activities recognized as such, and *true positives* or *hits* (H) are attacks correctly detected. The total number of attacks contained in an audit trail is given by $T = F_N + H$.

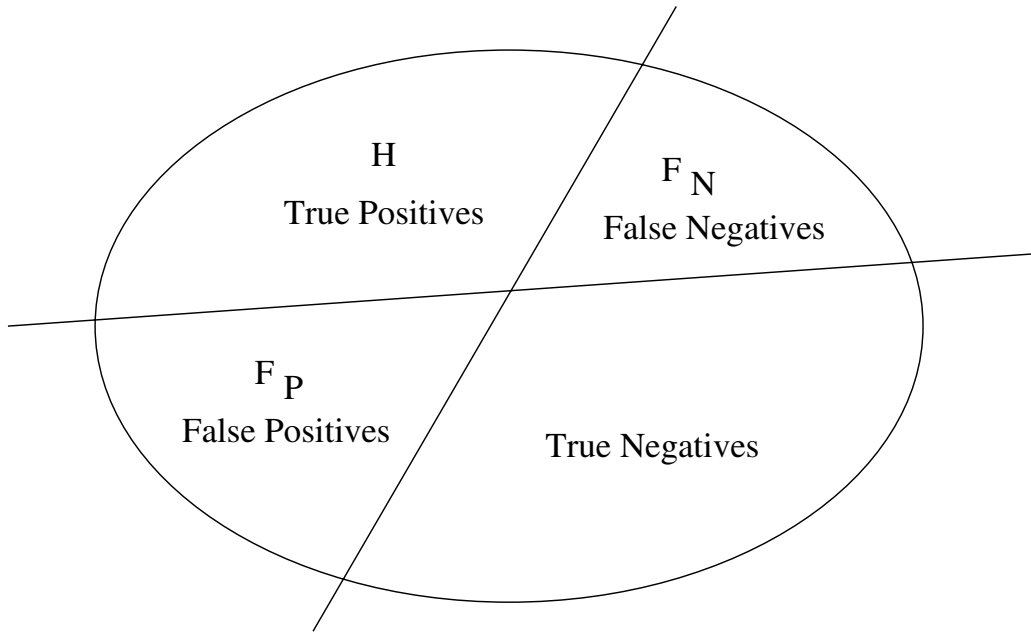


Figure 2.4: Classification of hits and misses in an IDS.

Analyzing the performance of an IDS requires evaluating different metrics. It is common practice to consider [64, 17]:

1. *Completeness*. It measures the ability to detect every malicious activity. It can be more formally be defined as follows:

$$Completeness = \frac{H}{T} = 1 - \frac{F_N}{H + F_N}$$

It gives a measure of how many false negatives are produced by the system. Completeness cannot be computed without complete knowledge of the entire audit trail.

2. *Accuracy*. It measures the ability to flag as attacks only the malicious activity. It can be more formally defined as:

$$Accuracy = \frac{H}{H + F_P} = 1 - \frac{F_P}{H + F_P}$$

It measures the number of false positives the system produces. Accuracy can be computed without complete knowledge of the entire dataset.

3. *Detection performance.* It measures the overall detection capability of the IDS, measuring both the number of false positives and false negatives produced. It is simply given by:

$$\begin{aligned} \text{Detection performance} &= \text{Completeness} \times \text{Accuracy} = \\ &= \frac{H}{T} \times \frac{H}{H + F_P} \end{aligned}$$

4. *Processing performance.* It measures the rate at which audit events are processed. It can simply be computed in terms of number of events processed per second.
5. *Timeliness.* It measures how fast the system can propagate the results of its analysis. A lower bound to it is given by the processing performance. More delay is added to perform the actual propagation of the results of processing.
6. *Fault tolerance.* This measures the ability of the IDS to defend itself from attacks, in particular denial-of-service attacks. There are no simple ways to produce an actual measure of this ability.

It is important to note that evaluating the overall performance of an IDS requires to take in account all of these metrics. Considering only one of them can lead to wrong assessments. For example, a system may flag every activity as malicious and achieve optimal completeness, but, of course, accuracy would be an issue. Analogously, if it considers all activities as legitimate, it produces no false positive and thus has optimal accuracy, but completeness would be low.

It is worth mentioning an analysis techniques used to correlate completeness and accuracy rate: it is the analysis of Receiver Operating Characteristic (ROC) curve. It is obtained by drawing the detection rate as a function of the false alarm rate. The IDS, at least in theory, can operate at any point on the curve, so the operator can vary the detection rate adjusting the false alarm rate. A typical ROC curve is shown in Figure 2.5. Also note that because it is usually difficult to define the false alarm rate, often the x axis shows false alarms per unit time.

As we discussed above, evaluation and comparison of IDSs is complicated by the need to account different facets of their performances. This, however,

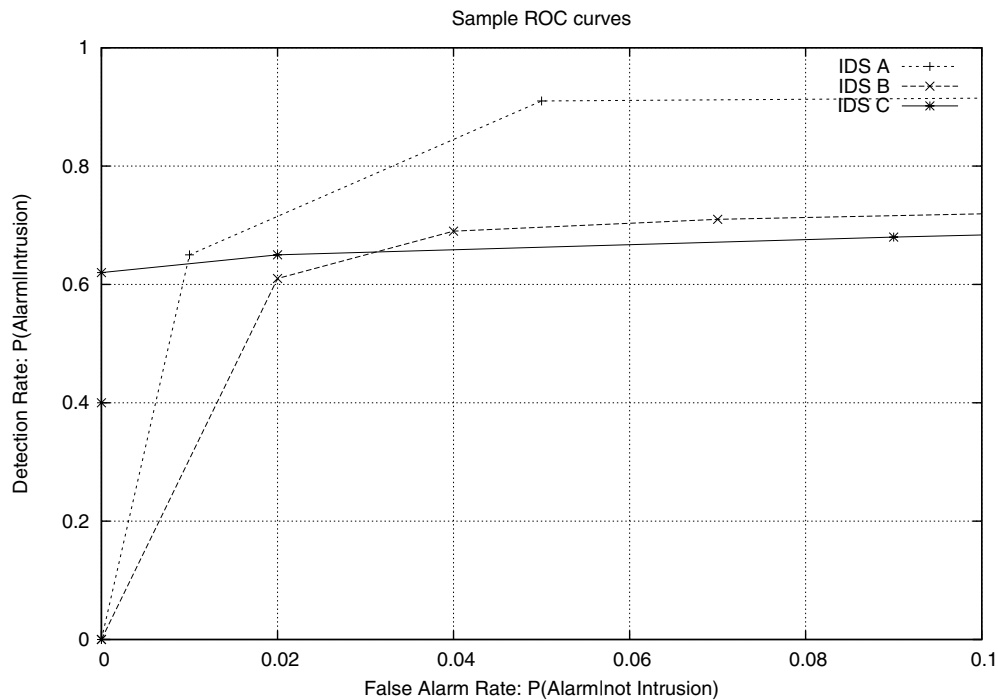


Figure 2.5: Sample ROC curves for three different IDSs.

is not the only challenge faced by IDS testers [58]. Other important aspects are:

- It is difficult to collect attack scripts and victim software. While exploits are often publicly released on the Internet and there exist websites and mailing lists devoted to discuss them, e.g., [11], it is time consuming to find scripts that actually work in the specific environment at hand. Similarly, it is sometimes difficult to obtain a version of the victim program that is vulnerable to the attack [41].
- Anomaly- and knowledge-based IDSs have different requirements, so it is difficult for a testing methodology to work well for both of them. In particular, anomaly-based systems need to be trained against a set of data that does not contain attacks. It is not always simple to obtain such data. Knowledge-based systems, on the other hand, are able to detect only attacks that are present in their knowledge base. The choice of which attacks include in the testing data set is pivotal in determining the performances of such systems. Different choices can lead to highly different results, compromising the repeatability of the testing process.

- Network- and host-based IDSs have different requirements. Testing host-based IDSs is usually more difficult because different systems use different input event streams. Also, host-based IDSs are often designed to work in real-time, and real-time tests are difficult to reproduce consistently.
- Testing dataset issues. It is generally recognized that using real datasets, comprising both of attacks and of normal background activities, is the most comprehensive way of testing IDSs. But privacy concerns often make this approach not viable. Furthermore, it is difficult to guarantee that the background activities do not contain some attack, and this interferes with the accuracy evaluation of systems. Approaches that do not use background data fail to test how an IDS performs in terms of false positives generation. They also assume that the performance of the IDS is not affected by the traffic rate, but it has been proved that in many cases the performance degrades, often significantly, if the traffic rate is high. Sometimes synthetic data is used, but creating such a simulation is costly and difficult. Also, it should be adapted to mimic different environments, e.g., activities at a e-commerce site are likely to be different to those of an academic network. A last approach consists in using real data. In most cases, these data need to be sanitized. This is a long and error prone process, e.g., it might accidentally end up removing important content or releasing sensitive information. Whatever is the approach chosen, the testing dataset should be well balanced so that it does not contain anomalies that advantage one IDS over the others.

Only a few large testing efforts have been performed to date. Some of the most developed efforts have been done by institutions as the University of California at Davis (1997) [68], MIT Lincoln Lab (1998 and 1999) [51, 52, 57], and Air Force Research Laboratory (1998) [21]. An overview of these testing efforts can also be found in [58]. The importance of these works resides not only in the identification of strengths and weak points of tested programs but also in the establishment of testing procedures and design and creation of testing tools.

2.5 Open Issues

Even though the field of intrusion detection has evolved rapidly in the last years, still some important issues need to be explored [40].

2.5.1 System Effectiveness

System effectiveness must be improved. The challenge is to develop a system that is able to detect all the attacks and produce a minimal number of false positives. Misuse based systems produce few false positives but don't detect unknown attacks. Behavior based systems are able to detect new attacks but typically produce many false positives. There are some projects trying to tie the advantages of the two approaches, building a hybrid system, e.g. [60] adopts both a statistical anomaly-detection unit and a signature-based inference unit.

2.5.2 Audit Throughput

New challenges are posed, especially to network-based IDS, by advances in technology. High-speed networks are commonplace and, thus, the amount of information the system needs to analyze can become unmanageable.

Some solutions have been proposed to face this problem: deploy many sensors in different peripheral locations of the network so that each of them has to deal with a limited amount of the traffic. The problem with this approach is that it is not always easy to decide where each sensor should be placed and, in addition, when an optimal placement is found, the deployment configuration has to dynamically keep up with network modifications.

Another possible solution consists in partitioning the stream of raw events so that its dimensions are reduced enough to be manageable. Of course, the partitioning must be done in a way that guarantees attack detection, i.e., slicing of the event stream must not separate information needed to detect an attack [44].

2.5.3 IDS Cooperation and Alert Correlation

A recent trend in intrusion detection is to use at the same time different IDS systems and correlate the analysis results and alerts. This approach aims at achieving higher-level descriptions of attacks or a more condensed view of the security issues highlighted during the analysis.

A number of requirements and possible application scenarios for IDS

inter-operation have been described [39]. For example, two IDSs might co-operate:

- Analyzing each other's generated alerts. This would be especially useful if they use different detection techniques, e.g., one is anomaly- and the other is misuse-based.
- Complementing each other's coverage, e.g., two IDSs, both host-based and located on two different hosts involved in an attack, or a network-based IDS and a host-based IDS, may produce an aggregate report on the malicious activity.
- Reinforcing each other's alerts to keep low the number of false positives generated.

IDSs use a number of languages to allow communication among their different components [82]:

- *Event Languages* are used to describe events and are used to communicate them between E-boxes and A-boxes;
- *Response Languages* are used to specify countermeasures against detected attacks;
- *Reporting Languages* are used to report alerts;
- *Correlation Languages* are used to specify relationships among attacks;
- *Exploit Languages* are used in knowledge based IDS to model attacks;
- *Detection Languages* provide the conceptual and operational framework for identifying attacks.

Detection and correlation languages relate to the internal workings of A-boxes. These languages have a key role in differentiating among systems: systems that use different detection and/or correlation languages usually show different characteristics and have specific strengths and weak points. Different approaches to intrusion detection often need different languages in order to reach their goals. Adopting one standard language would probably harness the ability to create new systems rather than creating the possibility for new solutions.

On the other hand, standardization of the other languages would be beneficial to the cooperation of different systems. Most of the work to date has been devoted to standardization of reporting language. The most influential proposal has been the Common Intrusion Specification Language

(CISL) [25], developed as part of the CIDF standardization effort. This language, based on S-expressions, provides a common reporting format and encoding for reports generated by IDSs. The CIDF effort has been taken over by the Internet Engineering Task Force (IETF) Intrusion Detection Working Group (IDWG)'s Intrusion Detection Message Exchange Format (IDMEF) effort [28]. Ideas explored by CISL have been included in the IDMEF format, that defines an XML-based message standard to report alerts and communicate sensor heart beats. The IDMEF format is now used in many IDS tools.

The adoption of the IDMEF format paved the way for a number of aggregation and correlation techniques and tools, e.g., [81, 18, 14, 15].

An initial experimental assessment of alert correlators has been described in [29]. A number of correlators have been tested under different metrics, e.g., attack-recognition, target identification capabilities and data reduction. The results are promising, but more explorations, in more realistic environment, are to be expected in the future.

Capitolo 3

STAT

In this chapter, an overview of the *STAT Framework* is presented. STAT is a framework for developing intrusion detection systems.

The STAT framework comprises four concepts:

1. The *STAT technique* is a method to model computer attacks in a high-level, abstract way.
2. The *STATL Language* is an extensible language used to represent attacks according to the STAT technique.
3. The *STAT Core* represents the runtime of the STATL language.
4. The *MetaSTAT* infrastructure allows STAT-based sensors to exchange alert messages and control directives in a distributed fashion.

In the following sections, we will analyze STAT's components, features, and applications.

3.1 STAT Technique

The *State Transition Analysis Technique* [37] is a method to describe computer penetrations as attack scenarios. An attack scenario models an attack as a series of *states* and *transitions* between states. States represent snapshots of the security relevant properties of the system. An attack scenario comprises at least two states: an initial, safe state and a final compromised, state. An attack models the evolution of a system from the initial state to the compromised state, passing through some intermediate states. For a general discussion about state based detection, refer to Section 2.3.2.

In STAT, each state is characterized by *state assertions*, which are predicates that specify conditions that hold in that state. Transitions are characterized by *signature actions* and by *transition assertions*. They specify what actions must be executed (signature actions) and what conditions must hold (transition assertions) to trigger a transition between two system states. Attack scenarios can be described graphically by means of *state transition diagrams (STDs)*.

Figure 3.1 models a single-step attack. Two states are represented: `init` and `end`. A transition, `toFinal`, leads the system from state `init` to state `end`. The transition is triggered if and only if both the assertion of the transition (t_1) and the assertions of the `end` state (s_1, s_2) hold.

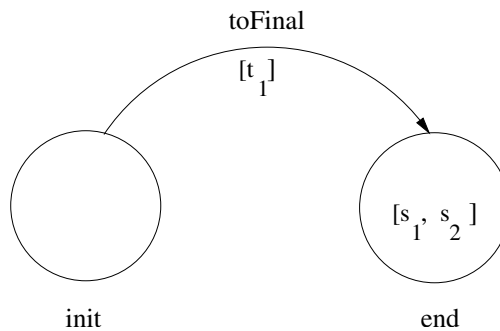


Figure 3.1: Sample State Transition Diagram.

More generally, let T be a transition between states A and B , with T 's transition assertions t_1, \dots, t_n and B 's assertions b_1, \dots, b_n . Then, if the current system state is A , the transition T is triggered if and only if:

1. $t_1 \wedge \dots \wedge t_n$ holds and
2. $b_1 \wedge \dots \wedge b_n$ holds.

For example, in a real-life attack scenario modeling an attempt to break the security of an operating systems, transition assertions would list commands the attacker has to launch and state assertions would specify properties such as file access control information, user ownership, etc.

The STAT approach has many significant features. It uses higher-level representations of attacks that are independent of the audit record format. This allows to use the same scenarios to detect attacks on different systems, characterized by different audit records. Therefore, attacks don't have to be re-modeled depending on the specific platform at hand. Also, this leads to attack models that are easier to understand, modify and maintain. For example, it is easier to understand the steps involved in an attack modeled as

a state-transition diagram rather than by examining the sequence of system calls issued by the attacker.

Another interesting characteristic of the STAT technique is that it allows to abstract from the details of the modeled attacks. In fact, it supports modeling attacks by representing only those steps that are critical for the effectiveness of the attack. In this way, it is possible to detect polymorphic attacks, i.e., attacks that can accomplish their goal by using slightly different methods.

Lastly, it allows to foresee an impending compromise and preempt or limit the attack before it is completed. Other approaches, for example the plain signature approach, only detects that an attempt to compromise the security of the system has happened. State transitions approaches, on the other hand, have the opportunity to see how the attack evolves, and possibly to stop it some steps away from its final goal.

3.2 STATL Language

STATL [22] is the language used to represent specifications of attack scenarios according to the STAT technique. In this section, we will briefly analyze its main characteristics.

3.2.1 Language Entities

The main concepts provided by the language are scenarios, states, transitions and timers.

Scenario

An attack scenario models an attack. It contains the states and transitions that compose the attack signature. Every scenario has a name and it may receive parameters. Libraries are used to introduce domain- or application-specific types, events, functions, and predicates.

The syntax of the scenario element is specified as follows:

```
Scenario ::=  
  { use LibraryID {',' LibraryID} ';' }  
  scenario ScenarioID  
  [ScenarioParameters]  
  '{'  
    [FrontMatter]  
    {State | Transition | NamedAction}
```

```
'}'
{ FunctionDefinition }
```

The following snippet of code defines a scenario named ftp_write:

```
scenario ftp_write
{
...
}
```

State

States are used to model the state of the system. Each state is identified by a unique name. It may have both assertions and code blocks. If present, the state assertion is tested before entry to the state, after testing the assertion of the transition that leads to the state. If it is not specified, it is implicitly true.

A state may include a code block, that specifies code to be executed after both the incoming transition's assertion and the state's assertion have been evaluated and found true and after the incoming transition's code block has been executed. The code block is typically used to maintain or modify state in local or global variables and to generate alerts.

Every scenario has at least two states: the initial state and a final state. The initial state has no incoming transition; every final state has no leaving transition.

The syntax of the state entity is as follows:

```
State ::=
  [initial]
  state StateId {Annotation}
  '{'
    [StateAssertion]
    [CodeBlock]
  '}'
```

In the following example, state *s1* is the initial state. State *s3* has an assertion, which checks whether the variable *counter* is greater than the value of the variable *threshold*, and a code block, which calls the built-in procedure *log* to write on disk the provided message:

```
initial state s1 { }

state s3
```

```

{
  counter > threshold
  {
    log{"In state s3"};
  }
}

```

Transition

Transitions model the transition of the system from one state to another. Each transition has a name and must indicate the leaving and entering state. Loops are allowed.

Each transition specifies a signature action. It may have both an assertion and a code block. Thus, a transition specifies what events to match and under what conditions. Transition's code blocks are typically used to copy event field values in the local or global environment for later reference.

Transitions can be *consuming*, *nonconsuming* or *unwinding*. A nonconsuming transition is used to represent a transition that does not prevent further occurrences of attacks to spawn from the source state. Both the leaving and entering state of the transition become valid. For example, if an attack has two steps that are the creation of link named `-i` to a SUID script and the execution of the script through the link, then the transition between the two steps is nonconsuming. In fact, after the second step, other executions of the script through the link may occur. Therefore, the first state is still valid.

On the contrary, a consuming transition makes the source state of an attack invalid. For example, if an attack requires to delete the `/etc/passwd` file, another occurrence of the attack cannot happen: the file cannot be deleted twice.

Unwinding transitions are used to represent conditions that invalidate the progress of an attack and to bring back the scenario to a previous state. For example, in a scenario that relies on the existence of a certain file, the deletion of that file makes the sequence of events non interesting. Therefore, the scenario may be brought back to a previous state, for instance before the creation of the file.

The syntax of the transition entity is defined as follows:

```

Transition ::=
  transition TransitionID '(' StateId '->' StateId ')'
  (consuming | nonconsuming | unwinding)
  {Annotation}

```

```
'{'
  ( '[' EventSpec ']' | ActionId)
  {Annotation}
  [ ':' Assertion]
  [CodeBlock]
}'
```

In the example below, a nonconsuming transition is shown. Its leaving state is *s0* and its entering state is *s1*. The signature action specifies a WRITE event. The transition assertion evaluates true if the WRITE event returned with success, the user that performed the action is not root and the newly created file is not in a directory owned by the user who triggered the event. The transition has a code block. The code is used to change the local environment:

```
transition create_file (s0 -> s1)
  nonconsuming
{
  [WRITE w] : (w.status >= 0) &&
              (w.euid != 0) &&
              (w.owner != w.auid)
  {
    inode = w.inode;
    objname = w.objname;
  }
}
```

Timer

Timers allow scenario writers to model attacks where a certain event must happen within an interval following some other event. For example, a simple scenario that detects port scans might state that if a request to a certain closed port is followed within, say, two minutes by another request to a different closed port, these events are part of a scan attempt.

In the following example, the code block of state *s1* starts timer *t1*, which will expire in 30 seconds. When it expires, the transition *expire* will fire:

```
scenario timer_example
{
  timer t1;

  state s1
```

```

{
  { timer_start(t1, 30); }
}

transition expire (s1 -> s2)
  consuming
{
  [ timer t1 ]
}
}

```

3.2.2 Language Extension

As we have seen so far, the STATL language provides constructs to model the domain-independent entities of an attack scenario. However, intrusion detection is performed in particular domains (e.g., networks, hosts) and environments (e.g., Linux, Windows). Therefore, intrusion detection developers must extend the language to match a particular domain or environment by adding definition of specific events, types, and predicates.

Events are defined as a set of C++ classes that represent the events in the specific audit stream to be analyzed. For example, a network-based IDS would define events representing UDP and TCP packets; a host-based IDS would define events associated with the reading and writing of files.

Predicates are added to test domain-specific properties. For example, a NIDS would need to test whether the source address of an IP packet matches the address of a specific server. A HIDS, on the other hand, would be interested in knowing the identity of the user who is reading or writing a certain file.

Extensions are packaged in a language extension module and compiled into a shared library. STATL scenarios access the events and predicates defined in an extension library by including it with the `use` keyword.

3.2.3 Translation

A number of tools are required to write a scenario and translate it in the form that will be used at run time. Figure 3.2, taken from [83], represents the various steps of this process. Shaded components are application-specific, meaning that are customized by different intrusion detection systems.

STATL attack scenarios can be created either directly in STATL, using any text editor, or in a graphical form using the *Scenario Editor* application.

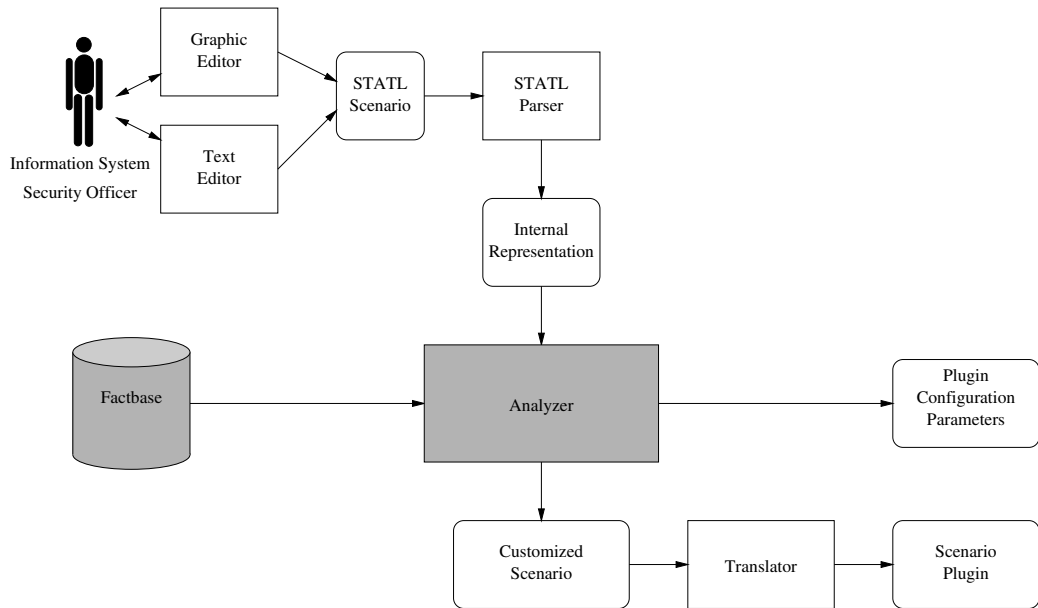


Figure 3.2: STATL translation process.

The Scenario Editor translates the graphical representation of the scenario into the correspondent STATL representation.

The *Parser* then reads the STATL source, checks that it is correct, and transforms it into an internal format that is used in the following steps.

The *Analyzer* reads a scenario in the common internal format and performs IDS-specific preprocessing before the scenario is translated into an executable representation. The STATL language provides *annotations* as a mean to extend the language with application-specific features. For example, a network-based intrusion detection system may use annotations to specify where a probe should be placed in the monitored network. Another component, the *Factbase*, is used to store information regarding the target environment, e.g., the topology of the target network. The analyzer process annotations and produces a version of the scenario possibly modified according to the information extracted from the Factbase.

Finally, the *Translator* performs the actual translation of the scenario, from the common internal format to C++ code. The C++ code is then compiled into a dynamically linked library (a “.so” file on Unix, or a “dll” on Windows) that is suitable to be linked in the runtime environment during execution.

3.2.4 Run Time Behavior

In order to understand the scenario execution semantics, that is the dynamic behavior of the elements of the STATL language, it is necessary to introduce some terminology.

A *snapshot* is defined as a 4-tuple containing a state name, a local environment, a local timer queue, and an event:

$$\sigma : N_S \times L \times Q \times E$$

where N_S denotes the set of state names, L is the set of local environments, E is the set of STAT events, and Q is the set of local timer queues. In other words, a snapshot represents the state of advancement of an attack that is evolved up to the state whose name equals the snapshot name. The snapshot event stores the event that led to the current state. The snapshot event also implicitly identifies *enabled* transitions of the snapshot σ . The enabled transitions in a set of transitions T with respect to snapshot σ are those whose source state is the same as the snapshot state.

An *instance* is a non-empty sequence of snapshots.

Finally, A STATL *scenario* is defined as a 6-tuple:

$$\mathcal{S} : (S, T, s_0, I, g, q)$$

where S is a set of states, T is a set of transitions, s_0 identifies the initial state, I is an instance set, g is a global environment, and q is a global timer queue. By representing a scenario instance as a sequence of snapshots, it is possible to represent the complete history of an attack modeled by a particular scenario \mathcal{S} . Each snapshot in an instance, other than the first, corresponds to a transition that fired in reaction to a particular event. The enabled transitions in a set of transitions T with respect to the instance i are the enabled transitions of the last snapshot of i .

During the scenario execution, only enabled transitions can be triggered. The semantics of transition triggering is the following:

1. Evaluate the transition assertion. If true, then
2. Evaluate the state assertion. If true, then
3. Execute the transition code block (this might modify the local and global environments) and then
4. Execute the state code block (this might modify the local and global environments).

Instances are created and destroyed as a consequence of the transition triggering mechanism that we have just described. In particular, when a nonconsuming transition is fired, a copy of the instance is created and a new snapshot is appended to the copy. The state of the new snapshot is the destination state of the transition that fired. The original instance is added to the new instance set. If the destination state is not the final state, the new instance is also added to the new instance set.

When a consuming transition is fired, a copy of the instance is created and a new snapshot is appended to the copy, exactly as in the case of non-consuming transitions. If the destination state is not the final state, the new instance is added to the new instance set. Note that the original instance is not added to the new instance set.

When an unwinding transition is fired, all the instances that are derived from the snapshot that is being unwound to are deleted (i.e., they are not in the new instance set).

A formal definition of scenario execution semantics is covered in more detail in [23].

3.2.5 Example

The following example is a slightly edited version of a scenario successfully used during the 1999 DARPA Intrusion Detection Evaluation.

```
use ustat;

scenario unix_ftp_write
{
  int user;
  int pid;
  int inode;
  string objname;

  initial state s0 { }

  transition create_file (s0 -> s1)
    nonconsuming
  {
    [WRITE w] : (w.status != -1) &&
                (w.euid != 0) &&
                (w.owner != w.auid)
  }
}
```

```

        inode = w.inode;
        objname = w.objname;
    }
}

state s1 { }

transition login (s1 -> s2)
    nonconsuming
{
    [EXECUTE e] : (e.status != -1) &&
                 match_name(e.objname, "login")
    {
        user = e.auid;
        pid = e.pid;
    }
}

state s2 { }

transition read_rhosts (s2 -> s3)
    consuming
{
    [READ r] : (r.status != -1) &&
              (r.pid == pid) &&
              (r.inode == inode)
}

state s3
{
    {
        string username;

        userid2name(user, username);
        log("remote user %s gained local access", username);
    }
}

transition logout (s2 -> s1)
    unwinding
{

```

```

    [EXIT e] : (e.pid == pid)
  }

  transition delete_file1 (s1 -> s0)
    unwinding
  {
    [DELETE d] : d.inode == inode
  }

  transition delete_file2 (s2-> s0)
    unwinding
  {
    [DELETE d] : d.inode == inode
  }
}

```

This scenario detects occurrences of the *ftp-write* attack. In this attack, the `ftp` service is used to create a `.rhost` file in the world-writable home directory of the “ftp” user. Using the created file, the attacker can open a session through the `rlogin` service without being required to provide a password.

The transition `create_file` is fired when a file has been created by a non-root user who doesn’t own the directory containing the file. The transition `login` signals the fact that a user has logged in. Lastly, the transition `read_rhosts` is triggered when that the suspicious file has been accessed by the login program. The first two transitions are nonconsuming: other attacks can be launched at those points of the original attack. The last transition is consuming because the attack is completed. There are also three unwinding transitions that bring back the system to previous steps when the suspicious file is deleted or if a user logs out. In case of the file deletion, an attack has to restart from the beginning, so the transition rolls back the scenario to state `s0`. When a user logs out, another attack can be launched without recreating the file `.rhost`, therefore the transition unwinds to state `s1`.

`READ`, `WRITE`, `EXECUTE`, `DELETE`, `EXIT` are abstractions of UNIX events. They are not part of the STATL language, but are provided as an extension by the `ustat` library (note the `use ustat;` instruction at the beginning of the scenario).

The state transition diagram that corresponds to this scenario is shown in Figure 3.3.

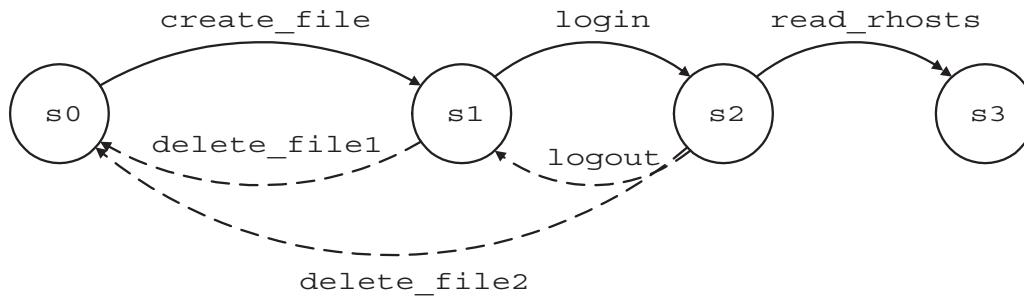


Figure 3.3: State Transition Diagram of `unix_ftp_write` scenario attack.

3.3 STAT Core

The STAT Core represents the runtime of the STATL language. The STAT Core implements the domain-independent characteristics of STATL, such as the concepts of state, transition, timer, matching of events, etc. At run-time the STAT Core performs the actual intrusion detection analysis process by matching an incoming stream of events against a number of attack scenarios.

The STAT Core must be extended in order to fit a specific environment. The resulting, specialized, intrusion detection tool is generically called a *STAT based sensor*.

A sensor is created by adding to the Core a number of modules: one or more language extension modules, event provider modules, scenario plugins and response libraries.

A *language extension module* describes the particular domain of the application and provides the domain-specific extensions to the STATL language, i.e., definition of new events, types, and predicates, as described in Section 3.2.2.

An *event provider module* collects raw events from the external environment, creates events as defined in a language extension, encapsulates these events into generic STAT events, and inserts the STAT events into the processing queue of the STAT Core. Note that many event providers could be active at the same time in one Core. For example, one provider may generate events by parsing system logs, while another one may generate events as the result of analyzing network traffic. Events of both types would be passed as input into the Core, thus enabling the Core to process scenarios that rely on both types of events. Event providers depend on one or more language extension modules, where event types are defined, and on a source of raw audit data, e.g., network sniffer, system log.

A *scenario plugin* is an executable representation of a STATL attack. The Core matches events received by the event provider with the regis-

tered scenarios. The details of scenario processing have been discussed in Section 3.2.4. Scenarios rely on language extensions for events, types, and predicates used in the scenarios.

Lastly, *response modules* are collections of functions that can be used to perform any type of response when an attack is detected.

Recalling the discussion about IDS general architecture in Section 2.2, it is worth observing that the event provider module has the role of an E-box; the Core loaded with language extension and scenario plugins covers the role of an A-box; the response modules correspond to an R-box. Finally, the set of scenario plugins loaded and activated in a Core constitutes the knowledge base of the sensor.

Table 3.1 summarizes dependencies and roles of the modules described above. *Activation dependencies* refer to dependencies that must be satisfied for a module to be loaded into the Core and run without failure. *Functional dependencies* indicate what components the module rely on to get its input. If functional dependencies are not satisfied the module is unable to perform any real work. Lastly, note that a module’s activation dependencies can be satisfied even when its functional dependencies are not.

<i>Module</i>	<i>Role</i>	<i>Activation deps.</i>	<i>Functional deps.</i>
Event Provider	E-box	Language Extension	Raw audit source
Scenario Plugin	Knowledge base	Language extension	Event Provider
Response Module	R-box	None	Scenario Plugins, External response devices
Language Extension	Domain specific modeling	None	None

Table 3.1: Dependencies and roles of STAT sensor modules.

3.4 Web of Sensors

In the previous Section we have examined what modules are necessary to configure the STAT Core and build a specialized sensor. It remains to analyze what mechanisms the STAT Framework offers to support this configuration process.

Most IDSs have limited functionalities to support dynamic sensor configuration. For example, many signature-based tools allow one to choose the set of signatures to be used for intrusion detection only at initialization time. But it may happen that the signature set needs to be updated, e.g., a new attack has been released and a signature for it has been developed, or an

existing signature has been rewritten in order to produce less false positives. Therefore, in such systems upgrading the signature set requires to stop the sensor, perform the actual update, and restart the sensor. More advanced configuration tweaks, e.g., adding a new audit event stream, are subject to even more constraints.

Furthermore, in many systems the configuration process is very-low level and has to be performed manually. In these cases, it can be very difficult to find the proper configuration for a sensor or even to check that the current configuration is appropriate for the specific sensor and the specific environment it operates in. Clearly, tools for automatic or semi-automatic and dynamic configuration are highly needed.

3.4.1 MetaSTAT

To overcome these problems, the *MetaSTAT* infrastructure has been developed [85]. MetaSTAT has been designed and implemented to support distributed intrusion detection, enabling dynamic reconfiguration and management of the deployed STAT-based IDSs. The idea is that detection is provided through a “web of sensors”, composed of distributed components linked together by a communication and control infrastructure. The configuration of a sensor can be modified in real-time to adjust it to deal with previously unknown attacks, changes in the security policy of the site, different level of concerns, etc.

MetaSTAT is comprised of several components, each responsible for different tasks. In the following paragraphs, we will present the MetaSTAT components.

Communication Components

CommSTAT provides the communication infrastructure that allows the sensors to exchange alert messages and control directives in a secure way. CommSTAT messages adhere to the IDMEF format [28]. The original format has been extended to include STAT-related control messages used to control and update STAT sensors. For example, messages to ship to a remote sensor a language extension and load it in the Core have been added. Other messages allow to transfer, load and activate all the other Core extension modules.

Sensors do not participate directly in the communication infrastructure. They are interfaced to it by a *STAT proxy*, that takes care of all the processing and messaging needs. This allows to build lightweight sensors and to integrate in this framework third-party tools that are not based on STAT.

Configuration and Management Components

The *Collector* component collects alerts generated by the deployed sensors and stores them in a relational database. A schema to efficiently store and retrieve IDMEF alerts have been developed, together with a GUI for the querying and display of stored alerts.

In order to support dynamic management and reconfiguration two additional components are used: the *Configurator* and the *Controller*. The Configurator maintains the *Module Database* and the *Sensor Database*. The Module Database keeps track of available STAT modules, i.e., event providers, language extensions, scenario plugins, and response libraries, and of their dependencies. The Sensor Database, contains the current configuration of the deployed sensors.

The Controller component allows one to send control messages to running sensors. Control messages are used to install or uninstall modules on a sensor, to dynamically load or unload modules in the sensor's Core, and to activate or deactivate modules.

Combining the information contained in the Module and Sensor Database and using the facilities offered by the Controller, it is possible to automatically determine and perform the steps required to change a sensor's configuration. For example, suppose that the Intrusion Detection Administrator (IDA) wants to re-shape the configuration of a sensor so that it can detect a particular attack against, say, the FTP service. This could be the consequence of the detection of scans for vulnerability in the FTP server. A suitable attack scenario is searched among the available scenarios or developed from scratch. The IDA orders this scenario to be activated on the sensor. Automatically, the scenario module is shipped to the sensor, all the dependencies of the scenario module are determined and the modules necessary to resolve them are also shipped. Finally, all the transferred modules are loaded and activated.

3.4.2 Configuration of a STAT Sensor

Figure 3.4, taken from [85], shows the steps involved in the dynamic configuration of a generic STAT based sensor through the MetaSTAT infrastructure, as described above. Advancement from one step to the following one is triggered by the reception of the appropriate control messages and extension modules. Figure 3.4 (a) shows a bare sensor, consisting of the only Core and Proxy. No intrusion detection functionality is provided at this point.

The first step in the configuration process consists in loading one or more event providers. It is done using appropriate MetaSTAT control messages. If

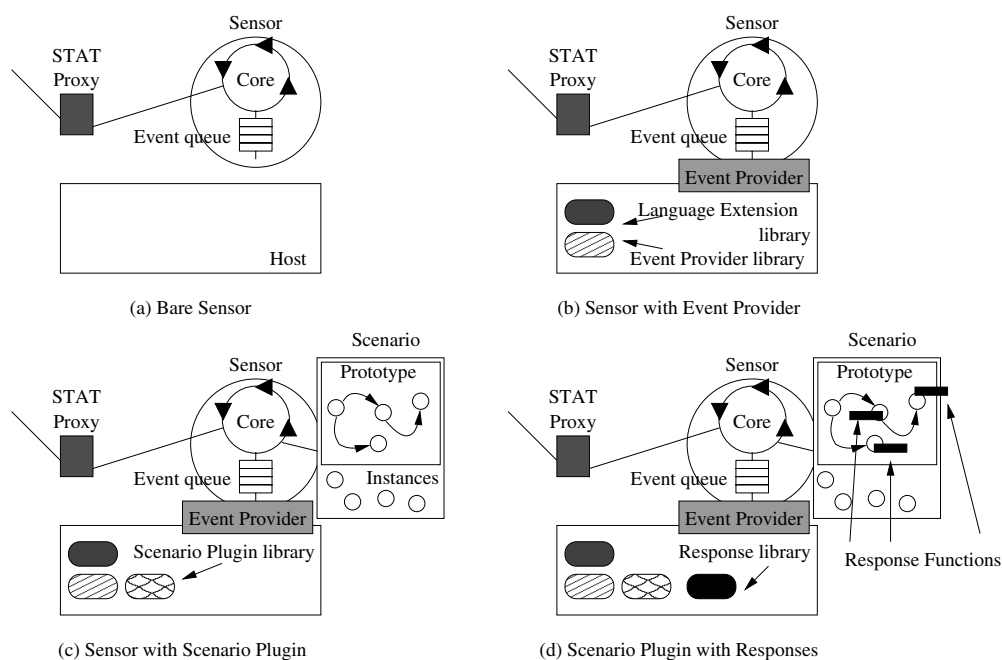


Figure 3.4: Configuration steps of a STAT based sensor.

the event providers rely on some language extension module, this is shipped and loaded as needed. At this point, the Core can process events received by the providers, but the attack knowledge base is still empty, so no intrusion can be detected. This configuration is shown in Figure 3.4 (b).

To start the detection process, it is necessary to populate the knowledge base, shipping scenario modules to the sensor and activating them. Again, if some dependencies are not satisfied, the necessary libraries are transferred to the sensor as well. This situation is depicted in Figure 3.4 (c).

Finally, Figure 3.4 (d) shows a fully functional sensor, that has received also a set of response extensions.

While we have examined how to control a single sensor, the STAT framework allows one to design, deploy and manage an IDS composed of highly distributed and interconnected components. The architecture of the overall IDS can then be thought of as a web of sensors, glued together and controlled by the MetaSTAT infrastructure. A visual representation of the resulting web of sensors is given in Figure 3.5, taken from [85].

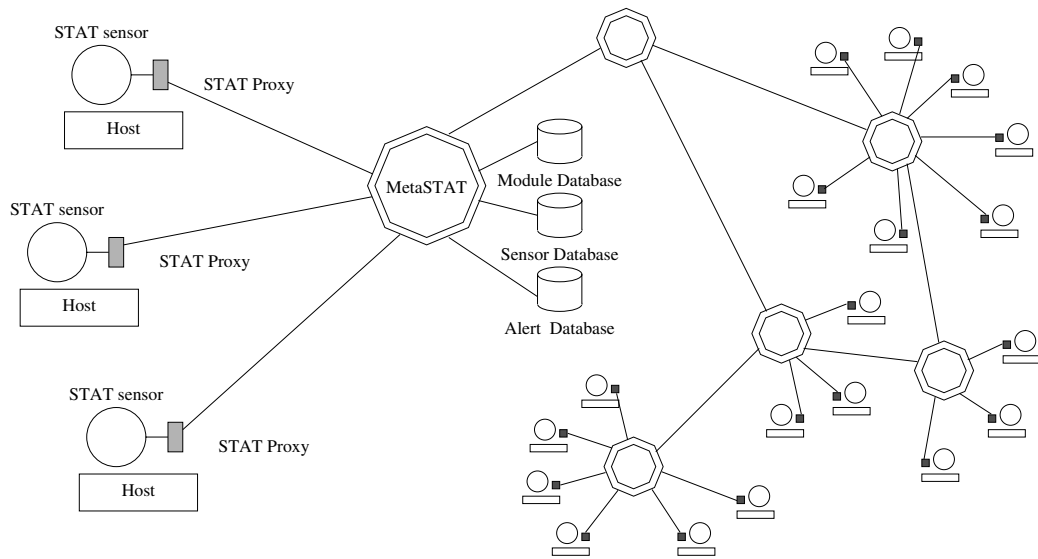


Figure 3.5: Architecture of a web of sensors.

3.5 Family of Sensors

The STAT Framework that has been presented in this chapter has been used to generate a suite of intrusion detection tools. For each tool in the suite, one or more extension modules containing application-specific event types and predicates was developed. In addition, a specific event provider was developed to translate the native audit records into abstract event representations. The tool suite currently includes *USTAT*, *WinSTAT*, *NetSTAT*, *WebSTAT*, *logSTAT*, *AODVSTAT*, *AgletSTAT*, *AlertSTAT* and *afedSTAT*. Of these, *NetSTAT* and *USTAT* were tested in the 1998 and 1999 DARPA Intrusion Detection System Evaluation efforts [21, 20].

3.5.1 USTAT

USTAT was the first application of the STAT technique to host-based intrusion detection. USTAT is a host-based IDS for the Solaris operating system [35, 36]. It uses the Sun Microsystems' BSM as a source of audit data. USTAT relies on two STATL extension library called *bsmlib* and *unixlib*. A number of scenarios have been developed to detect attacks specifically targeting Unix systems, such as buffer overflows and access to sensitive files by unprivileged applications.

3.5.2 WinSTAT

WinSTAT performs host-based intrusion detection in the Microsoft Windows NT/2000/XP environment. WinSTAT uses the event logs produced by the underlying operating system as input.

3.5.3 NetSTAT

NetSTAT is a network-based IDS [84]. A NetSTAT probe uses the network traffic sniffed from a network as input. It can also work offline by reading tcpdump files. NetSTAT relies on an extension library called *tcplib* that defines event types and predicates useful in the network environment. NetSTAT's event provider reassembles fragmented IP datagrams, reassembles TCP streams, parses DNS and RPC events and maintains the state of connections. Differently from most existing network-based intrusion detection systems, which are limited to the analysis of single packets, NetSTAT performs stateful analysis that takes into account the multi-step nature of some attacks.

3.5.4 WebSTAT and logSTAT

WebSTAT and logSTAT are two systems that perform intrusion detection at the application level. They both apply STAT analysis to the events contained in log files produced by applications. More precisely, WebSTAT analyzes the logs produced by Apache web server, and logSTAT parses UNIX syslog files.

3.5.5 AlertSTAT and afedSTAT

AlertSTAT is an alert correlator. It fuses, aggregates and correlates alerts from other intrusion detection systems. Scenarios have been written for AlertSTAT that identify complex, multi-step attacks. A typical example of such multi-step attack scenarios is the following. An attacker performs a port scan against a set of hosts, and he/she is detected by a network-based IDS. Secondly, the attacker gains access to one host performing a buffer overflow against a web server, and is detected by the application-level IDS, e.g., WebSTAT. Finally, an alert is generated by the host-based IDS on the victim host to indicate that the web server process has accessed a protected file on the local machine. The alerts generated by the single intrusion detection systems are correlated by AlertSTAT and a resulting alert is produced that conveys a much higher-level view of the overall attack

process. AlertSTAT operated on events formatted according to the IDMEF format.

Another correlator, called afedSTAT, has been developed. It uses events contained in a database of alerts, called AFED, which was developed by the Air Force Research Labs. The event provider of afedSTAT reads events from the database and translates them into IDMEF format events. As a consequence, it was possible to use all the scenarios defined for AlertSTAT.

3.5.6 AODVSTAT and AgletSTAT

The STAT Framework has been used also to develop intrusion detection systems that protect less traditional environments. AODVSTAT is an IDS that interprets AODV protocol messages and detects attacks against ad hoc wireless networks.

AgletSTAT is an IDS that analyzes the events generated by a mobile agent system, called *Aglets* [48], and detects attacks that exploit mobile agents.

Capitolo 4

LinSTAT

This chapter presents LinSTAT, a STAT-based sensor for Linux systems.

The LinSTAT intrusion detection system was developed leveraging the STAT Framework and adopting the methodology described in the previous chapter.

In the following sections, we will examine the reasons behind the development of an IDS targeted to the Linux platform and we will give an overview of the LinSTAT tool. We will present the LinSTAT audit data source in Chapter 5 and in Chapter 6 we will look in detail at how LinSTAT extends the STAT Core. A discussion of the developed attack scenarios is postponed to Chapter 7.

4.1 Overview

The adoption of Linux among common users and for mission-critical tasks makes it a valuable target for attackers.

Existing tools are able to address these security concerns only partially. With few exceptions, most of them are limited to the analysis of syslog files or user login/logout record or to integrity check of filesystems. Table 4.1 shows some of these tools and their features.

Especially in the field of host-based systems, there is a lack of full featured intrusion detection tools. The goal of our research is to develop mechanisms, techniques, and tools to support host-based intrusion detection in the Linux platform. The approach includes the implementation of mechanisms for the collection of complete auditing information and the development of an intrusion detection system that uses the collected audit trails to detect attacks.

<i>Tool Name</i>	<i>Reference</i>	<i>Functionality</i>
SWATCH	[31]	Log file analysis based on regular expressions
LogSentry	[53]	Log file analysis based on keyword
Prelude	[65]	HIDS based on log file analysis
HostSentry	[34]	Login anomaly detection
Snort	[71]	NIDS
Tripwire	[42]	File system integrity checker
RPM	[72]	Package integrity checker

Table 4.1: Intrusion detection tools for Linux

4.2 Architecture

Being LinSTAT a STAT-based sensor, its architecture is dictated by the STAT Framework. LinSTAT is built around the STAT Core, extended with a Linux-specific language extension, an event provider and a number of attack scenarios. Figure 4.1 shows the architecture of the system.

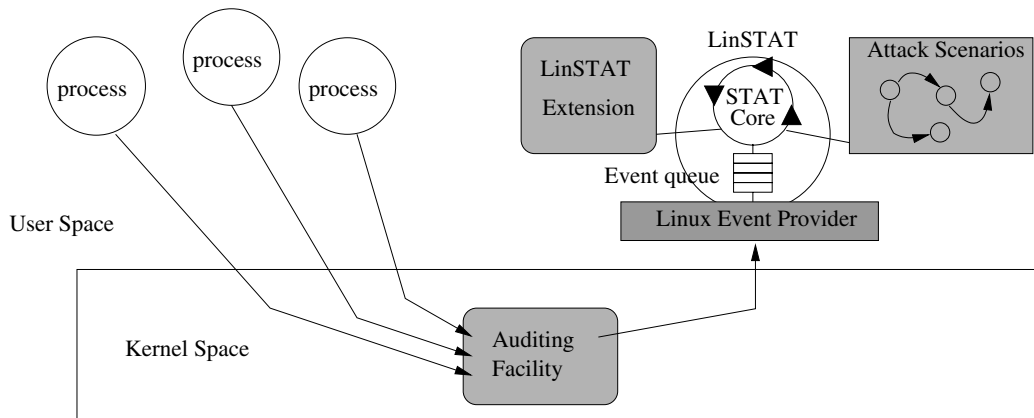


Figure 4.1: LinSTAT architecture

The kernel is instrumented with an auditing facility that monitors process activities and makes available to user-space the collected audit information. The event provider gathers this raw audit data, filters it, and transforms raw audit events into STAT events. A set of language extensions and attack scenarios allows the generic STAT Core to interpret the event stream and match it against the modeled attacks. Although it is not shown in figure for the sake of clarity, every LinSTAT sensor can be attached to the MetaSTAT infrastructure and be remotely configured and controlled. The components shown in shaded color are LinSTAT-specific.

Figure 4.2 shows a diagram that describes the logic followed by LinSTAT in processing an event stream. LinSTAT can work either in offline or real-time mode. In the first case, events are read from a file. If no event is available, it means the event file has been completely processed and LinSTAT exits. In real-time mode, the event stream is obtained directly from the kernel. If no event is available at the time of reading, it means that the current process' activity is not producing events. LinSTAT is put to sleep by the kernel and automatically waken up when events are again available for processing. The LinSTAT provider analyzes the first available event and uses it to create a new STAT event. The STAT event is passed to the Core that adds the event into the Event Queue. A Core thread is responsible for dequeuing events from the Event Queue, updating scenario instances according to the new event and performing the actual intrusion detection by matching scenario instances and modeled attacks. If an attack occurrence is detected, a response is triggered. In the figure the generation of an alert is shown, but more sophisticated and proactive responses are also possible.

The general characteristics of LinSTAT are summarized in Table 4.2.

<i>Characteristic type</i>	<i>LinSTAT</i>
Detection method	Stateful analysis: State transition
Behavior on detection	Active
Audit source location	Host-based: C2 audit trail
Usage frequency	Continuous

Table 4.2: Characteristics of LinSTAT

LinSTAT is a host-based sensor. It analyzes events produced by a kernel-level audit facility. LinSTAT inherits from the Core component a stateful intrusion detection engine. State transition analysis is performed according to the semantics of the STATL language. When an intrusion is detected, the default action is to simply generate an IDMEF formatted alert. This allows to use the output generated by LinSTAT as input of an aggregator and correlator system, e.g., AlertSTAT. By simply loading and activating a suitable response library in the LinSTAT sensor, it is possible to produce active reaction to contrast detected attacks. The typical usage mode is real-time monitoring. However, it is also possible to analyze event data in an off-line fashion, processing static files containing an audit trail.

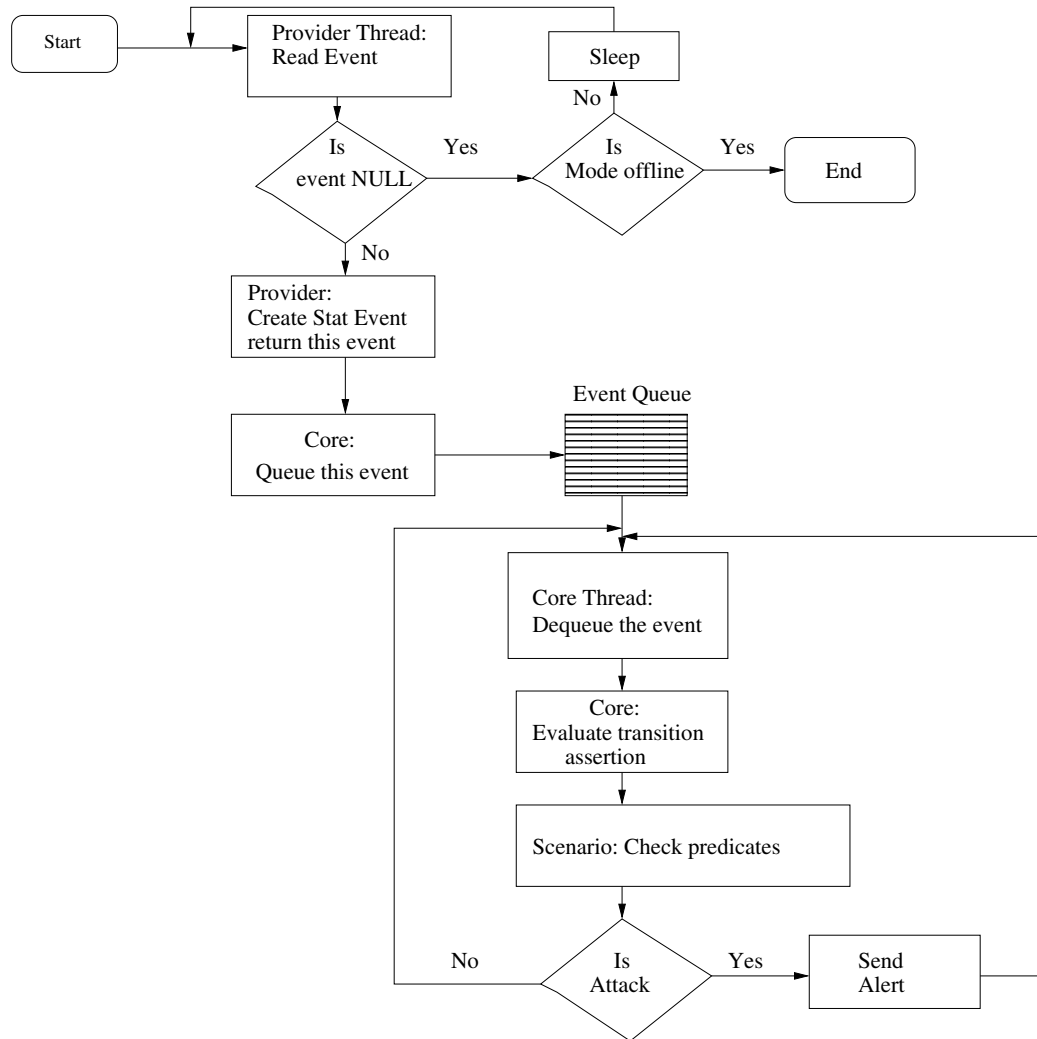


Figure 4.2: LinSTAT state machine diagram.

Capitolo 5

LinSTAT audit Data Source

Given the general architecture of LinSTAT as sketched in the previous chapter, it is clear that the quality of intrusion detection depends heavily on the quality of the data provided for analysis. Ideally, a data source for intrusion detection should provide enough data to detect every attack that leaves a trace on the monitored system. Furthermore, it should have the following properties [13]:

1. **Flexibility:** the auditing system should be easy to configure to conform to different auditing levels.
2. **Timeliness:** audit data should be delivered without significant delay to the analyzer module.
3. **Performance:** the system should impose minimal degradation of system throughput.
4. **Correctness:** each event must be semantically and syntactically correct. Furthermore, the order of events must be preserved in the audit trail.

In this chapter we describe the audit data source used in LinSTAT and how it complies with these requirements.

5.1 LinSTAT Audit Source

In this section we describe the details of the LinSTAT audit source design and implementation.

5.1.1 What Type of Monitoring

The first design choice we had to take was whether the system had to perform direct or indirect monitoring [75]. Direct monitoring means that the system obtains data directly from the objects that generate it. On the contrary, in indirect monitoring the data is obtained from a source, usually a log file, that reflects the behavior of the object to be monitored.

We chose direct monitoring because this approach has a number of advantages. In particular, it gives access to data that in the indirect monitoring is usually not available, e.g., internal operations of a program that are usually not recorded in a log file. Furthermore, data produced by indirect monitoring can be more easily altered by an intruder before the intrusion detection system has a chance to use it.

5.1.2 What to Monitor

Secondly, we had to choose what operations we wanted to monitor. Our goal is to detect attacks against the whole system, not just a few programs. Therefore, we decided to collect audit data associated with system call invocations.

This approach is based on a few fundamental assumptions: (1) the system state is initially safe and (2) every program that wants to modify the system state, e.g., to lead it to a unsafe state in the case of an attack, has to interact with the system itself through the system call mechanism. Note that with this decision we disregard other events, like interrupts and exceptions, that can modify the system behavior. As a consequence, this means that LinSTAT is unable to detect an attack that does not use system calls. For example, if a flaw in the divide-by-zero exception handler could be used to gain root access to a system, an attack could be performed that does not invoke system calls and, thus, that does not leave any manifestation in the system call audit trail. However, based on operating system history, attacks like these seem to be very unlikely to occur.

5.1.3 Where to Monitor

At this point, one more design option is available: it is possible to trace system call invocations and collect audit data either in user-space or in kernel-space.

The kernel-space approach requires to modify the kernel in order to add audit capabilities. By working in kernel-space, one can have access to the content of critical data structures, like the ones describing processes, open

files, and socket descriptors. It is possible to inspect parameters passed to system calls and to examine/modify the flow of execution. Furthermore, because the access to this information is immediate, this method promises to be effective under a performance point of view. Another advantage is that kernel-level auditing, cannot be easily disabled or modified by an intruder. However, it has some disadvantages with respect to both development and deployment. In particular, coding, testing, and debugging kernel code is usually a more difficult process than the development of user-space code and the resulting artifact is usually specific to a certain kernel version. A notable example of this approach is the BSM facility, available on Solaris systems [78].

The user-space solution requires to wrap the actual system call invocations into library functions. These library functions implement the audit data collection mechanism. The main advantage of this approach is that it makes easy to modify, add, or remove the auditing facility from a host. However, it has disadvantages: it can be easily tampered with or disabled by an intruder, and the wrapping mechanism is usually expensive in terms of performance. Lastly, not all the information needed can be readily available. Examples of this method have been described in [46] and [38].

We chose to follow the kernel-space approach, instrumenting the Linux kernel to add auditing capabilities.

5.1.4 Implementation

Our implementation is based on the `auditmodule` tool of the SNARE package [73].

`auditmodule` is a loadable kernel module designed as a set of functions that wrap the code of the original system calls in order to gather audit information. The collected data is then provided via the `/proc` filesystem to user-space programs.

The original module audits file-related system calls (`open`, `creat`, `mkdir`, `unlink`, `mknod`, `rmdir`, `chown`, `chmod`, `symlink`, `link`, `rename`, `truncate`), process-handling system calls (`execve`, `exit`), privilege-setting system calls (the `setuid`-like functions), networking system calls (`connect` and `accept`) and other system calls (`reboot`, `chroot`, `create_module`). The collected information includes the calling process (process id, command line invocation, parent process id), the user owning the process (user id, group id, effective user id, and effective group id), the completion time of the system call, the return value, and other information that can be directly obtained from the system call parameters (e.g., the path of the executed program for `execve`, source and destination path for `rename`, destination address for `connect`, and

source address for `accept`).

The original `auditmodule` tool has some limitations that impair its ability to provide complete and useful audit data to an IDS. In particular, some important events are not monitored, e.g., mounting and unmounting a filesystem or removing kernel module, and only limited information about files is collected, e.g., the i-node number is not provided. This allows an attacker to perform potentially hostile actions without being detected, e.g., mounting a filesystem that should not be accessed, and to adopt evasion techniques based, for instance, on symlinks and hardlinks.

Because of its open-source nature, it was possible to extend the original `auditmodule` to fit our needs. We implemented the auditing of some system calls that have particular relevance to system security (`mount` and `umount`, `delete_module`) and the monitoring of process creation (`fork`, `vfork`, and `clone`). For every system call that involves files, we added the auditing of meta-information for the file: mode bits, owner uid, owner gid, file type, device, i-node number. The audit data now contains the full path name of the executable that issued the system call. With respect to the `connect` and `accept` functions, the data audited has been extended to include both source and destination addresses. Lastly, particular attention has been put in trying to avoid the typical limitations of audit data collection systems described in [66].

It is important to observe that, because of the modular design of a STAT based sensor, it is possible to use a different data source changing only a minimal portion of the existing code (approximately 6.5%).

To complement the `auditmodule`, we have written a user-space program, called `auditdaemon`, that reads the audit data and writes it in binary format to a file. This is used for offline analysis, when audit data needs to be collected for later analysis. We also developed a program, `praudit`¹, that prints the audit information in a human-readable format.

5.2 Audit Data

Every time a system call is invoked, the `auditmodule` creates one or more audit events and inserts them in the audit trail.

Every audit event is described by a set of tokens. Some of them are common to all events: action type, time when the system call returned, value returned, command line invocation of the program that issued the system call, pathname of the program, `uid` (user id), `gid` (group id), `euid`

¹“`praudit`” is also the name of the BSM audit record printer for the Solaris OS.

(effective user id), `egid` (effective group id), `pid` (process id), `ppid` (parent process id). Other tokens are event specific, e.g., an audit trail of a system call involving files contains also the tokens `objname` (pathname of the file), `owner` and `gowner` (user and group id of the file's owner), `inode` (i-node of the file), `dev` (the device id of the device where the file is stored), and `perm` (permission bits).

As regard to the implementation, the token set is serialized in a binary format. While other formats (e.g., XML) would allow for easier parsing, it was critical to minimize the size of data in order to optimize both the time needed to move events from kernel- to user-space and the space used to store events in memory and on disk. Figure 5.1 shows the relationship between the data transfer cost and the amount of data transferred. The figure has been obtained benchmarking the number of cycles required by the kernel function `copy_to_user` to move data from kernel- to user-space. As can be seen, the cost of moving data across the kernel- and user-space boundary is linearly proportional to the amount of data transferred.

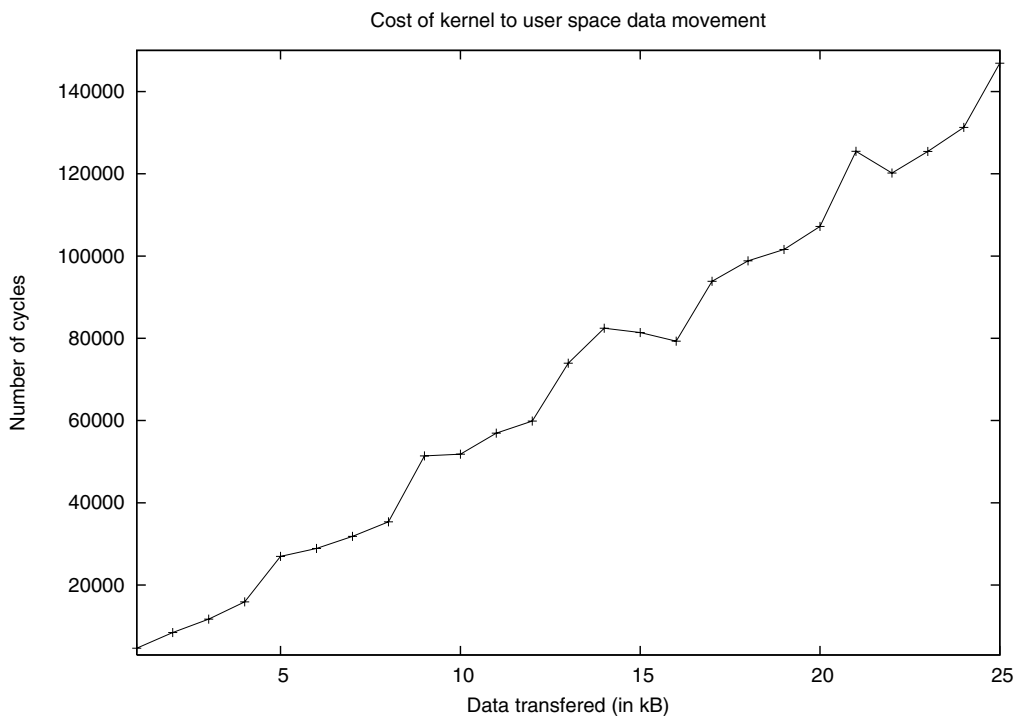


Figure 5.1: Cost of data transfer from kernel to user-space.

Furthermore, because Linux is used on different platforms, particular care, especially regarding byte ordering, has been taken in order to make

the system portable.

A sample event as shown by `praudit` is presented in Figure 5.2. The audit

```
act: EXECUTE, time: Wed Aug 14 18:10:37 2002, retcode: 0,  
exec_args: cat /proc/devices, pathname: /bin/bash, uid: 0,  
gid: 0, euid:0, egid: 0, pid: 13921, ppid: 13920,  
pwd: /home/mcova, objname: /bin/cat, owner: 0, gowner: 0,  
inode: 3817535, dev: 774, perm: rwxr-xr-x
```

Figure 5.2: An example of an event entry in the audit trail.

record represents an EXECUTE action. More precisely, the root user reads the file `/proc/devices` through the program `cat`. The `act` token is used to specify the action type. The `time` token specifies when the action happened and, `retcode` is used to store the return value of the action. In this case, the return value is 0, meaning that the system call was completed successfully. The field `pathname` is the full canonicalized pathname of the program that triggered the audited action. In this case it is the shell interpreter: the action happened during an interactive shell session. `exec_args` specifies the command line invocation of the executed program. The tokens `uid`, `gid`, `euid`, and `egid` convey information on the user running the program, in this case the root user. `pid` and `ppid` are used to specify the process id and parent process id of the running process, while `pwd` is the working directory of the process. `objname` is used to specify the path of the file that is the target of the action. In this case, `objname` contains the pathname of the program executed (`/bin/cat`). The following tokens contain various pieces of information about the file specified by `objname`: the file's owner id is stored in `owner`, the file's group owner id in `gowner`, the file's i-node is stored in `inode`, the device id of the device where the file is located is stored in `dev`. Finally, the permission bits are specified by `perm`.

5.2.1 Raw Audit Events

The system call set monitored by LinSTAT can be partitioned in a number of audit classes. System calls in the same class have similar scope, e.g., modifying files or changing user's identity, and contain the same type of auditing information. Table 5.1 lists the audited system calls, the class they are in, and a brief explanation of the class itself.

<i>System call</i>	<i>Audit class</i>	<i>Class description</i>
open creat mkdir unlink mknod rmdir chmod truncate chroot truncate64	IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS IO_CLASS	I/O related syscalls
execve	EXECUTE_CLASS	Process execution
chown lchown chown32 lchown32 fchown	CHANGE_OWNER_CLASS CHANGE_OWNER_CLASS CHANGE_OWNER_CLASS CHANGE_OWNER_CLASS CHANGE_OWNER_CLASS	Change owner syscalls
symlink link rename mount umount oldumount	DOUBLE_PATH_CLASS DOUBLE_PATH_CLASS DOUBLE_PATH_CLASS DOUBLE_PATH_CLASS DOUBLE_PATH_CLASS	Syscalls involving 2 files
reboot exit	PROCESS_CONTROL_CLASS PROCESS_CONTROL_CLASS	Process control syscalls
setuid setreuid setresuid setuid32 setreuid32 setresuid32 setgid setregid setresgid setgid32 setregid32 setresgid32	SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS SUBSTITUTE_CLASS	Identity substitution syscalls

socketcall	NET_CLASS	Networking syscalls
create_module delete_module	ADMIN_CLASS ADMIN_CLASS	Administration syscalls
fork vfork clone	PROCESS_SPAWN_CLASS PROCESS_SPAWN_CLASS PROCESS_SPAWN_CLASS	Process creation syscalls

Table 5.1: System calls audited by LinSTAT.

5.2.2 Audit Information

In this section we provide details about what information is collected for each audit class event.

Before delving into the details of LinSTAT audit data, we review some of the UNIX concepts necessary to better understand the information provided to the intrusion detection engine by audit events. A good reference on the UNIX system is [76].

Every UNIX process has associated with it a real user and group ID, and an effective user and group ID. Other IDs are used by the operating system, but are not used by LinSTAT. The real user ID and real group ID identify the identity of the user who started the process. These two IDs are extracted from the user entry in the password file. Specific system calls exist to retrieve and modify the value of these fields. The effective user ID and effective group ID determine the process' file access permission. When a user executes a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. It is possible, however, setting a special flag in the program file's mode word, to cause the effective user ID to be the user ID of the owner of the file, and, similarly, the effective group ID to be the group owner of the file.

Every UNIX process is uniquely identified by a numerical process ID. Furthermore, every process knows the parent process ID, i.e., the process ID of the process it originated from.

To complete this brief discussion about UNIX processes, we mention that every process has a working directory. This is the directory from which all relative pathnames are interpreted.

Every file in the system has associated with it a user and group owner ID. These are used to determine the owner of the file and the access permission of processes to the file. Every file also has a mode word, which specifies what categories of users and what type of accesses are permitted to the file.

The i-node is a data structure used to maintain information about a file. In particular, the i-node contains the owner of the file, the size of the file, the device it is located on, and pointers to where the actual data blocks for the file are located on disk.

Lastly, it is worth mentioning that system calls return a value greater than or equal to 0 to indicate successful completion. A negative value, instead, is used in case of errors.

IO_CLASS

Table 5.2 shows the content of an audit event for actions in the IO_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
pwd	working directory of the process performing the action
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action
objname	full pathname of the file involved in the I/O operation
owner	owner ID of the owner of 'objname'
gowner	group ID of the owner of 'objname'
inode	i-node of objname
dev	device ID of the device where 'objname' is located
perm	permission bits of 'objname'

Table 5.2: Audit data for I/O events.

The information contained in a IO_CLASS event allows one to determine the user that requests the IO operation, the process used to complete it, and the file involved in this operation. For example, the following lines present an IO_CLASS event as shown by `praudit`:

```
act: READ, time: Wed Sep 10 16:10:04 2003, retcode: 3,
exec_args: cat, pathname: /bin/cat, uid: 500, gid: 500, euid:
```

```
500, egid: 500, pid: 1187, ppid: 1159, pwd: /home/marco,
objname: /etc/ld.so.cache, owner: 0, gowner: 0, inode: 229405,
dev: 773, perm: rw-r--r--
```

The trace shows that the program `/bin/cat` opened for reading the file whose path is `/etc/ld.so.cache`. The program was running with process ID 1187 on behalf of the user whose ID is 500. The operation was successful as indicated by the positive return value. Additional information regarding the time this action was performed and the read file is also provided in the trace, and can be interpreted according to Table 5.2.

EXECUTE_CLASS

Table 5.3 shows the structure of an audit event for actions in the EXECUTE_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
pwd	working directory of the process performing the action
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action
objname	full pathname of the file executed
owner	owner ID of the owner of 'objname'
gowner	group ID of the owner of 'objname'
inode	i-node of 'objname'
dev	device ID of the device where 'objname' is located
perm	permission bits of 'objname'

Table 5.3: Audit data for I/O events.

Every time a user requests to execute a program, the information contained in a EXECUTE_CLASS event allows one to determine the user identity, the program whose execution is requested, and the program used to run

the new process. As an example, the following audit trail excerpt presents an EXECUTE_CLASS event:

```
act: EXECUTE, time: Wed Sep 10 16:10:04 2003, retcode: 0,
exec_args: cat getifaddrs.c, pathname: /bin/bash, uid: 500,
gid: 500, euid: 500, egid: 500, pid: 1187, ppid: 1159, pwd:
/home/marco, objname: /bin/cat, owner: 0, gowner: 0, inode:
3457063, dev: 773, perm: rwxr-xr-x
```

The trace shows that the program `/bin/bash`, running with process ID 1187, executed the program `/bin/cat`. The user that requested the execution has ID 500. For a complete interpretation of all the fields shown in the trace, refer to Table 5.3.

CHANGE_OWNER_CLASS

Table 5.4 shows the content of an audit event for actions in the CHANGE_OWNER_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
pwd	working directory of the process performing the action
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action
objname	full pathname of the file involved in the operation
owner	new owner user ID of 'objname'
gowner	new owner group ID of 'objname'
inode	i-node of 'objname'
dev	device ID of the device where 'objname' is located
perm	permission bits of 'objname'

Table 5.4: Audit data for CHANGE_OWNER events.

The information contained in a CHANGE_OWNER_CLASS event allows one to determine the user that wants to modify the ownership of a file, the

file whose ownership is to be changed, and the process used to complete the operation. The following lines present a `CHANGE_OWNER_CLASS` event as shown by `praudit`:

```
act: MODIFY_OWNER, time: Wed Sep 10 16:58:23 2003, retcode: 0,
exec_args: chown, pathname: /bin/chown, uid: 0, gid: 0,
euid: 0, egid: 0, pid: 1310, ppid: 1307, pwd: /home/marco,
objname: /home/marco/getifaddrs.c, owner: 99, gowner: 99,
inode: 9044495, dev: 773, perm: rw-r--r--
```

The trace shows that the user whose ID is 0 changed the owner of the file `/home/marco/getifaddrs.c`. The new owner is set to be the user with ID 99. The remaining fields of the event are as explained in Table 5.4.

DOUBLE_PATH_CLASS

Table 5.5 shows the content of an audit event for actions in the `DOUBLE_PATH_CLASS` class.

A sample `DOUBLE_PATH_CLASS` event is shown to clarify the information contained in this event class:

```
act: SYMLINK, time: Wed Sep 10 16:16:31 2003, retcode: 0,
exec_args: ln, pathname: /bin/ln, uid: 500, gid: 500,
euid: 500, egid: 500, pid: 1230, ppid: 1159, pwd: /home/marco,
objname: /home/marco/getifaddrs.c, owner: 500, gowner: 500,
inode: 9044495, dev: 773, perm: rw-rw-r--, target:
link_to_getifaddrs.c
```

The trace shows that the user with ID 500 created a symbolic link to the file `/home/marco/getifaddrs.c`, using the program `/bin/ln`. The link was created with name `link_to_getifaddrs.c`. The remaining fields of the event are set as explained in Table 5.5.

PROCESS_CONTROL_CLASS

Table 5.6 shows the content of an audit event generated in correspondence of actions in the `PROCESS_CONTROL_CLASS` class.

The following event trace exemplifies the information contained in a `PROCESS_CONTROL_CLASS` event:

```
act: EXIT, time: Wed Sep 10 16:16:31 2003, retcode: 0,
exec_args: ln, pathname: /bin/ln, uid: 500, gid: 500, euid:
500, egid: 500, pid: 1230, ppid: 1159
```

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
pwd	working directory of the process performing the action
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action
objname	full pathname of the “first” file involved in the action (the order is the same as in the command line invocation of programs such as ln, rename, mount, umount, i.e., the file being linked to, renamed, the (u)mounted device)
owner	owner ID of the owner of ‘objname’
gowner	group ID of the owner of ‘objname’
inode	i-node of ‘objname’
dev	device ID of the device where ‘objname’ is located
target	pathname of the “second” file (the order is the same as in the command line invocation of programs such as ln, rename, mount, umount, i.e., the name of the link, the new name of the file being renamed, the mount point)
perm	permission bits of ‘objname’

Table 5.5: Audit data for DOUBLE_PATH_CLASS events.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action

Table 5.6: Audit data for PROCESS_CONTROL_CLASS events.

The trace shows that the process with ID 1230, executing the program `/bin/ln` on behalf of the user with ID 500, exited. Additional information is available as indicated in Table 5.6.

SUBSTITUTE_CLASS

Table 5.7 shows the content of an audit event for actions in the SUBSTITUTE_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action

Table 5.7: Audit data for SUBSTITUTE_CLASS events.

SUBSTITUTE_CLASS events are triggered when a process requires to substitute its original identity with a new one. The information contained

in such events allows one to determine the process that wants to modify its identity, the user that owns the process, and the identity the process wants to obtain. As an example, the following audit trail excerpt presents a typical `SUBSTITUTE_CLASS` event:

```
act: CHANGE_IDENTITY, time: Wed Sep 10 16:58:23 2003, retcode:
0, exec_args: su, pathname: /bin/su, uid: 0, gid: 0, euid: 500,
egid: 500, pid: 1308, ppid: 1307
```

The process with process ID 1308, executing the program `/bin/su`, wants to set its user ID to 0. The effective user ID of the process is still set to 500. Note that if also the effective user ID is changed, the original identity of the process cannot be obtained from this event alone. However, when this information is necessary, it can be retrieved by examining the user ID field of any previous event referring to the process with ID 1308.

PROCESS_SPAWN_CLASS

Table 5.8 shows the structure of an audit event for actions in the `PROCESS_SPAWN_CLASS` class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the newly created process
ppid	process ID of the process performing the action
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall

Table 5.8: Audit data for `PROCESS_SPAWN_CLASS` events.

The following event trace exemplifies the information contained in a `PROCESS_SPAWN_CLASS` event:

```
act: CREATE_PROCESS, time: Wed Sep 10 16:58:21 2003, retcode:
1307, exec_args: bash, pathname: /bin/bash, uid: 500, gid: 500,
euid: 500, egid: 500, pid: 1307, ppid: 1159
```

The trace shows that the process with ID 1159, executing the program with path `/bin/bash` created a new process, identified by ID 1307. Other fields of the trace are set as explained in Table 5.8.

NET_CLASS

Table 5.9 shows the structure of an audit event for actions in the NET_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
src_ipaddr	network information about the source of the connection (IP address and port)
dst_ipaddr	network information about the destination of the connection (IP address and port)
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	first 15 characters of command line invocation
pathname	full pathname of the process performing the action

Table 5.9: Audit data for NET_CLASS events.

The information contained in a NET_CLASS event allows one to determine the source and destination node of a network connection. In particular the IP addresses and the port numbers are provided. NET_CLASS events are generated in correspondence of invocations of the `connect` and `accept` system calls. Therefore, they mainly keep track of TCP connections. Note, however, that also UDP socket can issue a `connect` system call. The following trace exemplifies a NET_CLASS event:

```
act: ACCEPT, time: Tue Aug 13 22:30:00 2002, retcode: 4,  
exec_args: sshd, pathname: /usr/sbin/sshd, uid: 0, gid: 0,  
euid: 0, egid: 0, pid: 680, ppid: 1, source addr:  
128.111.48.123:32805, dest addr: 0.0.0.0:22
```

The event shows that the process with ID 680, executing the program with path `/usr/sbin/sshd` (the secure shell daemon), accepted a connection on the local machine and port 22. The node that originated the connection has IP address 128.111.48.123. The port used by the remote machine to establish the connection has number 32805. The remaining fields of the event can be interpreted referring to Table 5.9.

ADMIN_CLASS

Table 5.10 shows the structure of an audit event for actions in the ADMIN_CLASS class.

<i>Field name</i>	<i>Field content</i>
uid	real uid of the process performing the action
gid	real gid of the process performing the action
euid	effective uid of the process performing the action
egid	effective gid of the process performing the action
pid	process ID of the process performing the action
ppid	process ID of the parent process
act	the action type
time	time when the action occurred
retcode	return code of the corresponding syscall
exec_args	command line invocation
pathname	full pathname of the process performing the action
objname	name of the module being installed

Table 5.10: Audit data for ADMIN_CLASS events.

The following trace exemplifies the information provided by an ADMIN_CLASS event:

```
act: DELETE_MODULE, time: Wed Sep 10 17:55:37 2003, retcode: 0,  
exec_args: insmod.old, pathname: /sbin/insmod.old, uid: 0,  
gid: 0, euid: 0, egid: 0, pid: 1471, ppid: 1427, objname: vfat
```

The user with ID 0 removed the kernel module `vfat`, using the program with path `/sbin/insmod.old`. Other fields are set according to Table 5.10.

Capitolo 6

LinSTAT Extensions

In this chapter we will examine the LinSTAT-specific extensions to the Core. We will discuss LinSTAT events, LinSTAT types, the LinSTAT Factbase, and LinSTAT predicates.

6.1 LinSTAT Events

The LinSTAT event provider is responsible for receiving raw audit data from the audit source and transforming it into a stream of events that the Core can analyze. In order to achieve this task, the provider parses the raw audit stream, extracts information about each single raw event, and generates one or more `LINUX_Events` that model the raw event. `LINUX_Event` is a subtype of `STATExtEvent`, which, in turn, implements the concept of event in the STAT Core.

A `LINUX_Event` models an event in terms of subjects, actions, and objects¹. A subject is the initiator of an action and an object is an entity involved in the action. In our approach, an action is the invocation of a system call. Users and processes are identified as subjects, and files as objects. It is important to observe that Linux, being a UNIX-like operating system, regards as files all the available resources, from networking cards to memory and hard disks. Therefore the choice of limiting the set of objects to the set of files of the system does not impact the generality of our approach.

Each event provides the Core with the following information about subjects: their real user and group id, their effective real and group id, their process and parent process id, the working directory and the pathname of the program that was used to perform the action. If the process has been

¹The identification of subjects, actions and objects as fundamental components of every intrusion detection system was first presented in [19].

generated on behalf of a remote user, the local and remote address is stored. This set of information allows one to identify what process is responsible for a certain action and the user that ran the process.

Actions are identified by the action type that specifies what kind of action is described in the event. The time the action was executed, whether it was successful or not, is also stored in the event. Note that we often refer to a specific “even type” meaning an event whose action type coincides with the specified one.

Lastly, to identify objects, an event provides their filename, user and group ownership, i-node and device number, and information regarding permission and other file flags.

Table 6.1 lists the event types introduced by LinSTAT and their meaning. These events can be used in signature actions of Linux-based scenarios.

<i>Event Type</i>	<i>Meaning</i>
READ	A subject wants to read a file
WRITE	A subject wants to write a file
CREATE	A subject wants to create a file
DELETE	A subject wants to delete a file
EXECUTE	A subject wants to execute a program
EXIT	A process wants to terminate its execution
MODIFY_OWNER	A subject wants to change the ownership of a file
MODIFY_PERM	A subject wants to change the permissions of a file
RENAME	A subject wants to rename a file
HARDLINK	A subject wants to create a hardlink to a file
SYMLINK	A subject wants to create a symlink to a file
READ_CREATE	A subject wants to read a file or create it if nonexistent
READ_WRITE	A subject wants to read and write a file
WRITE_CREATE	A subject wants to write a file or create it if nonexistent
READ_WRITE_CREATE	A subject wants to read and write a file or create it if nonexistent
CHANGE_IDENTITY	A subject wants to change his/her identity
INSTALL_MODULE	A subject wants to install a new kernel module
DELETE_MODULE	A subject wants to remove a loaded kernel module
MOUNT	A subject wants to mount a new filesystem
UMOUNT	A subject wants to unmount a filesystem
CONNECT	A subject wants to connect to a remote host
ACCEPT	A subject wants to accept a connection from a remote host
CREATE_PROCESS	A subject wants to create a new process

Table 6.1: LinSTAT event types.

Figure 6.1 represents a `LINUX_Event`, showing in detail its relationship with the `STATExtEvent` class and the information it carries. Every class modeling a domain-specific STAT event derives from the `STATExtEvent` class.

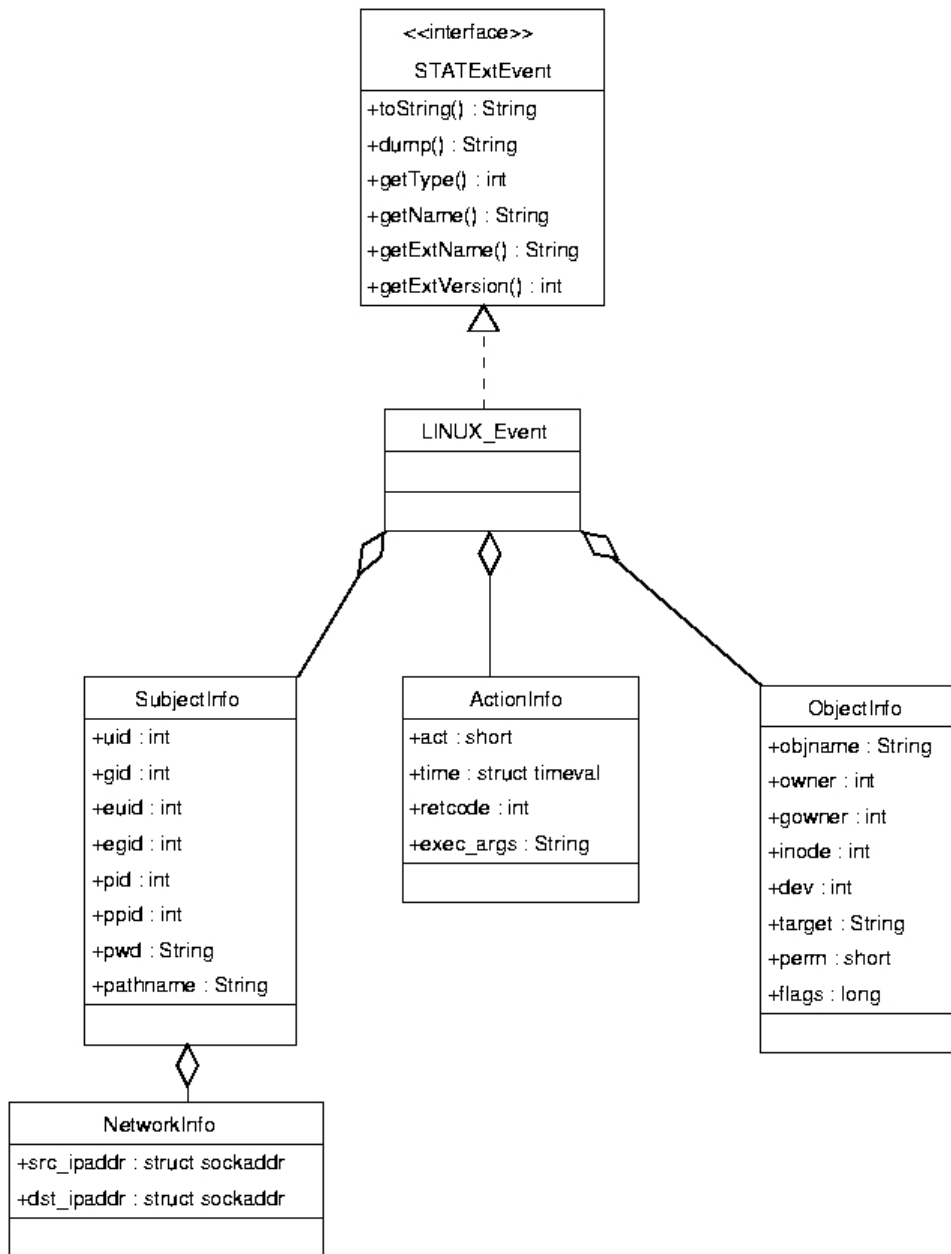


Figure 6.1: UML diagram of LINUX_Event.

The `STATExtEvent` class provides a number of methods that serve as an interface with the STAT Core and that are inherited by the subclasses. Therefore, the subclasses define only the structure of domain-specific events and don't have to worry about the details of interacting with the Core. This design makes the process of developing new event types easier and less error-prone, because it shields away the complexity of the Core. Also, it allows to change the way the STAT Core handles events, without the need to modify events' implementation. The `LINUX_Event` class defines the structure of a Linux-specific event in terms of subjects, actions, and objects. The content of a `LINUX_Event` object is public accessible. While a design based on the getter/setter methods pattern would probably have been cleaner and more adherent to the principle of information hiding, the cost of these additional method invocations would have had a not negligible impact on system performance.

Having presented the new STAT events that are introduced by LinSTAT, it remains to examine how raw audit events are translated into STAT events. Table 6.2 shows the correspondence between raw audit events and STAT events. Note that different system call invocations can be modeled by the same STAT event and, also, that system calls with different parameters can be modeled by different STAT events.

6.2 LinSTAT Factbase

Every STAT sensor is provided with a *Factbase*, i.e., a collection of information structures for encoding knowledge regarding the local environment. LinSTAT's Factbase consists of filesets, each of which lists files that share some characteristics that make them vulnerable to certain attacks or that require specific monitoring.

Each fileset is uniquely identified by a unique name and contains a list of file or directory names, a list of programs, a list of users and a list of groups. The relationship between files, users and program is defined in scenarios, not in the fileset itself. The structure of a fileset is:

```
FILESET id
  OBJECTS {
    list of file
  }
  PROGRAMS {
    list of programs
  }
  USERS {
```

<i>STAT Event Type</i>	<i>System calls</i>
READ	open
WRITE	open, truncate, ftruncate, ftruncate64, truncate64
CREATE	open
DELETE	unlink, rmdir
EXECUTE	execve
EXIT	exit, reboot
MODIFY_OWNER	chowner, lchown, chown32, lchown32, fchown
MODIFY_PERM	chmod, fchmod
RENAME	rename
HARDLINK	link
SYMLINK	symlink
READ_CREATE	open
READ_WRITE	open
WRITE_CREATE	open, creat, mknod, mkdir
READ_WRITE_CREATE	open
CHANGE_IDENTITY	setuid, seteuid, setresuid, setuid32, seteuid32, setresuid32, setgid, setregid, setresgid, setgid32, setregid32, setresgid32
INSTALL_MODULE	create_module
DELETE_MODULE	delete_module
MOUNT	mount
UMOUNT	umount, umount2
CONNECT	socketcall
ACCEPT	socketcall
CREATE_PROCESS	fork, vfork, clone

Table 6.2: Correspondence between raw audit events and STAT events.

```

    list of users (numerical id)
}
GROUPS {
    list of groups (numerical id)
}

```

Each sublist/section (OBJECTS, PROGRAMS, USERS, GROUPS) may contain any number of elements. If a section is missing, it is to be considered empty. It is allowed to use the following limited form of regular expression to specify items in each list: ‘prefix*’ will match any item that starts with the string ‘prefix’.

Four filesets are predefined and share the same semantics in all STAT modules. Additional filesets may be defined to take care of domain, platform, or installation specific issues, e.g., USTAT and LinSTAT define filesets that relate to mail and printer spool directories that are not defined in WinSTAT. These are the standard filesets and their intended use:

- **RESTRICTED_WRITE_SETUP_FILES**: the OBJECTS section lists files that should be written to only through programs listed in the PROGRAMS list. USERS (GROUPS) is a list of users (groups) that are allowed to access objects in OBJECTS with programs that are not in PROGRAMS. This fileset is used by scenarios that detect modification of system configuration files. The USERS and GROUPS sections are left empty in the default configuration.
- **RESTRICTED_READ_SETUP_FILES**: the OBJECTS section lists files which should be accessed only through programs listed in the PROGRAMS section. USERS (GROUPS) is a list of users (groups) that are allowed to access an object in OBJECTS with programs that are not in PROGRAMS. This fileset is used by scenarios that detect unauthorized reading of system configuration files. The USERS and GROUPS sections are left empty in the default configuration.
- **RESTRICTED_WRITE_DIRECTORIES**: the OBJECTS section lists directories where new files should be added only through programs listed in PROGRAMS. This fileset is used by scenarios that detect implantation of Trojan horses in directories that are likely to be included in users’ PATH environment variable. As such, PROGRAMS, USERS, and GROUPS should be left empty.
- **RESTRICTED_WRITE_EXECUTABLE_FILES**: the OBJECTS section lists executable files that should be written to and/or deleted only using programs in PROGRAMS. USERS (GROUPS) is a list users (groups)

that are allowed to delete or modify files in OBJECTS with programs not in PROGRAMS. This fileset is used by scenarios to detect deletion of executable system file (DOS attack) or modification (Trojan horse implantation). USERS, GROUPS and PROGRAMS are empty by default.

STAT's Factbase may also contain an IGNORE section. The IGNORE section lists all pair of (object,action) that should be ignored by a STAT sensor. The format of this section is

```
IGNORE {  
    object action  
}
```

where object is any object in the system (file and directories) and action is an action signature, defined by the STAT sensor, performed on object. A special action is defined, ANY, that matches with any action.

6.3 LinSTAT Types and Predicates

Many of the types and predicates defined by LinSTAT are used to access and query the Factbase. For example, a number of predicates allow one to know if a specified entity is in one of the sections of the Factbase or if a pair (object,event) is to be ignored. Table 6.3 shows all these predicates.

Other predicates (see Table 6.4) are available to test whether a filename matches a given pattern.

Lastly, other predicates are available that don't fit in any of the previous categories (see Table 6.5).

6.4 The Tainting Mechanism

In this section we describe a new process tracking mechanism implemented by LinSTAT.

It is often useful to track processes that are created on behalf of a remote user and associate them with the IP addresses of the hosts the remote user is connected from. In this way, it possible to create and enforce different policies for local and remote users. For example, a user that is locally using a machine could be entitled to access some files, while these remain inaccessible to a remote user.

<i>Predicate</i>	<i>Meaning</i>
member (name, fileset_id)	returns true if 'name' is in the OBJECTS section of the fileset identified by 'fileset_id'
member_dir (name, fileset_id)	returns true if 'name' is in a directory specified in the OBJECTS section of the fileset identified by 'fileset_id'
member_dir_sub (name, fileset_id)	returns true if 'name' is in a directory or in a subdirectory of a directory specified in the OBJECTS section of the fileset identified by 'fileset_id'
in_prog_set (name, fileset_id)	returns true if 'name' is an item of the PROGRAMS section of the fileset identified by 'fileset_id'
in_user_set (uid, fileset_id)	returns true if uid is an item of the USERS section of the fileset identified by 'fileset_id'
in_group_set (gid, fileset_id)	returns true if gid is an item of the GROUPS section of the fileset identified by 'fileset_id'
ignore_this (name, action)	returns true if the pair ('name', 'action') is to be ignored

Table 6.3: Predicates used to query the Factbase.

<i>Predicate</i>	<i>Meaning</i>
match_prefix_path(fname, pattern)	Returns true if the filename 'fname' starts with 'pattern'
match_prefix_file(fname, pattern)	Returns true if 'pattern' is the base-name of the filename 'fname'
match_fullname(fname, pattern)	Returns true if the filename 'fname' matches the pattern 'pattern'
match_name(fname, pattern)	Returns true if the basename of the filename 'fname' matches the pattern 'pattern'
match_multiple_name(fname, pattern1, pattern2)	Returns true if the filename 'fname' contains, in order, the patterns 'pattern1' and 'pattern2'

Table 6.4: Predicates used to perform pattern matching on filenames.

<i>Predicate</i>	<i>Meaning</i>
<code>contains_non_ascii(s)</code>	Returns true if the string 's' contains non-ASCII characters
<code>shell_script(fname)</code>	Returns true if the file 'fname' is a shell script
<code>userid2name(uid)</code>	Returns the username corresponding to the user ID 'uid'

Table 6.5: Other LinSTAT predicates.

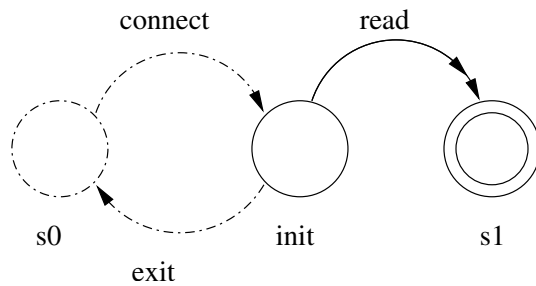
LinSTAT implements a mechanism to perform the described process/IP address association. This mechanism, called “network tainting”, marks processes generated by connections from remote machines. Every process that derives from a tainted one is also tainted. For every tainted process LinSTAT keeps track of where it originated from (source IP address and port) and where it connected to (destination IP address and port). Tainting information is kept in a dynamic hash table: whenever a new connection is created to the local machine, the process that accepted the connection is stored in the table and the relevant information is saved. As the process spawns new processes, new entries are added in the table. These are removed when the processes end their execution.

A scenario that takes advantage of the tainting mechanism is shown in Figure 6.2. This scenario generates an alert whenever a tainted process tries to read a secret file, namely, the configuration file of the ssh server. On the contrary, local users can access the file without generating an alert. Note that the alert will show the address and port the remote user is coming from and the address and port he/she connected to. Also note that in the scenario graphical representation the first state and transition are showed using a dash and dot line style. This is because the first transition is triggered automatically whenever a `connect` event is generated and thus need not be explicitly included in the scenario’s STATL description.

A number of predicates has been added to LinSTAT in order to get information about tainted processes, see Table 6.6.

<i>Predicate</i>	<i>Meaning</i>
isTainted(pid)	Returns true if the process whose ID is 'pid' is tainted
getSrcPort(pid)	If the process whose ID is 'pid' is tainted, returns the port information associated with the process
getSrcPortStr(pid)	If the process whose ID is 'pid' is tainted, returns a string representing the source port information associated with the process
getDstPort(pid)	If the process whose ID is 'pid' is tainted, returns the destination port information associated with the process
getDstPortStr(pid)	If the process whose ID is 'pid' is tainted, returns a string representing the destination port information associated with the process
getSrcAddr(pid)	If the process whose ID is 'pid' is tainted, returns (as a long) the source address information associated with the process
getDstAddr(pid)	If the process whose ID is 'pid' is tainted, returns (as a long) the destination address information associated with the process
getSrcAddrStr(pid)	If the process whose ID is 'pid' is tainted, returns the source address information associated with the process (in dotted decimal format)
getDstAddrStr(pid)	If the process whose ID is 'pid' is tainted, returns the destination address information associated with the process (in dotted decimal format)
networkDataToString(pid)	If the process whose ID is 'pid' is tainted, returns a string encoding the network information associated with the process (source address:source port → destination address:destination port)

Table 6.6: LinSTAT's tainting predicates.



```

use linux;

string secret_file = "/etc/ssh/sshd_config";

scenario linux_tainting_test
{
  int uid;
  string source_addr;
  unsigned short source_port;
  string dest_addr;
  unsigned short dest_port;

  initial state init { }

  transition read (init -> s1) nonconsuming
  {
    [READ r] :
    isTainted(r.pid, stat) &&
    match_fullname(r.objname, secret_file)
    {
      uid = r.uid;
      src_addr = getSrcAddrStr(r.pid, stat);

      src_port = getSrcPort(r.pid, stat);
      dest_addr = getDstAddrStr(r.pid, stat);
      dest_port = getDstPort(r.pid, stat);
    }
  }

  state s1 {
    { log ("%s read by tainted process (%s:%hd -> %s:%hd)",
          secret_file, src_addr, src_port, dest_addr,
          dest_port);
    }
  }
}

```

Figure 6.2: A LinSTAT attack scenario that uses the tainting mechanism.

Capitolo 7

Testing

The evaluation of the prototype implementation addresses both the auditing system and the intrusion detection tool. The auditing system has been evaluated quantitatively by determining the overhead introduced by the logging procedures. The intrusion detection system as a whole has been evaluated by quantifying the overhead of running it with a different number of scenarios loaded and activated. Lastly, we have evaluated it from the functional point of view by testing the developed scenarios and controlling that the alert generation performed as expected, and by running some publicly available exploits and checking whether the attacks were correctly detected.

7.1 Performance Testing

The goal of performance testing was to evaluate the overhead incurred running LinSTAT. Furthermore, we wanted to be able to identify how the total overhead was distributed among the different parts of the system.

7.1.1 Testing Methodology

A batch of jobs was issued and the elapsed time to complete the jobs recorded. Each test was repeated 10 times.

The following jobs were issued:

1. *Compile*: complete compilation of a Linux kernel, using make. This is a CPU and I/O intensive job.
2. *Matmul*: Matrix multiplication program. This job is CPU intensive.

In order to isolate the impact caused by each component to the overall architecture, each job was issued with the following configurations:

1. Both LinSTAT and the auditing system are not running. This configuration is used as a term of comparison for the following ones. We call it the *base configuration*.
2. The audit data source is installed and activated and the `auditdaemon` is writing audit data on the disk. This gives us the cost of running the audit facility, both the kernel and user-space part of it. Note that there is no intrusion detection capability with this configuration. This is the *auditdaemon configuration*.
3. LinSTAT is running with a number of scenarios loaded and activated increasing from 1 up to 14. The tainting mechanism is disabled. These configurations allow us to evaluate the total cost of performing intrusion detection and to predict how the system scales when the knowledge base size, i.e., the number of modeled attacks, increases.
4. The same configurations as in the previous case, but the tainting mechanism is activated and the jobs are run so that every new process created during the test is tainted. These configurations allow us to measure the impact of the tainting mechanism on the overall performance.

We also evaluated the memory requirements of LinSTAT. This was computed recording the virtual memory size of the process (as given by `ps -o vsize`).

The evaluation of LinSTAT was carried out on a system with a Pentium III CPU clocked at 800 MHz, with 264 MB of RAM, and an WDC 7200 rpm hard disk with a UDMA-33 interface. The operating system was Linux, kernel version 2.4.2 (RedHat 7.1 standard system), of course modified with our auditing module.

The Matmul Job

To test how LinSTAT performed together with a CPU bound process we used the `Matmul` program.

This program performs a number of matrix operations and only performs a limited amount of I/O, printing on the console the results of its computation. The amount of events generated by this program is very low, because only a small number of system calls are invoked. The memory usage is usually high, typically over 80%.

The Compile Job

To test how LinSTAT performed together with a CPU and I/O process we ran it while a complete recompilation of the Linux kernel was in act.

During the execution of this job almost 3,000 processes are created and more than 212,000 events are generated by the audit facility. This means that about 470 events per second have to be passed from the audit facility to the event provider and then to the STAT Core for analysis.

7.1.2 Results

The results of the Matmul test are shown in Figure 7.1 and 7.2. The results of the Compile test are shown in Figure 7.3 and 7.4.

For each job-configuration pair, except for the ones belonging to the base configuration, the percentage difference between the completion time and the value obtained for the base configuration is reported. The times are given in the minutes:seconds.milliseconds format and represent the average over 10 runs. For tests with the tainting mechanism activated, no results are shown for the `auditdaemon` configuration, because this configuration has no intrusion detection capability and thus the tainting mechanism is never used.

As expected, the auditing system's impact is more visible when it is run concurrently with processes that create a high number of events. CPU-bound processes generate only a few events and thus put a only limited stress on the system. Processes that frequently access disk or spawn new processes, instead, generate a high volume audit trace that put the system under a more significant test. The overall impact on performance is, however, promisingly limited.

7.1.3 Components of System Overhead

In order to understand the impact of different LinSTAT components on the total overhead, we will break up the discussion analyzing different processing steps, from the point where the audit data is generated in the kernel to the point where it is analyzed by the intrusion detection tool.

Kernel Auditing

The impact of instrumenting the kernel can be measured by comparing the second configuration to the base configuration. As the percentages show, it is less then 2.5%. This figure includes both the cost of gathering the audit information in kernel-space and logging it into disk. Tests conducted with the audit data source installed and activated but no logging activity showed

Matmul test

<i>Conf</i>	<i>time</i>	<i>Overhead</i>	<i>time taint</i>	<i>Overhead taint</i>
base	3:10.206	-	-	-
auditdaemon	3:11.953	0.92%	-	-
0 scen.	3:12.310	1.11%	3:13.827	1.90%
1 scen.	3:12.251	1.08%	3:14.023	2.01%
2 scen.	3:12.148	1.02%	3:13.912	1.95%
3 scen.	3:12.053	0.97%	3:14.190	2.09%
4 scen.	3:12.187	1.04%	3:14.264	2.13%
5 scen.	3:12.075	0.98%	3:14.463	2.24%
6 scen.	3:12.421	1.16%	3:14.649	2.34%
7 scen.	3:12.070	0.98%	3:14.670	2.35%
8 scen.	3:11.855	0.87%	3:14.791	2.41%
9 scen.	3:12.512	1.21%	3:14.793	2.41%
10 scen.	3:12.416	1.16%	3:14.887	2.46%
11 scen.	3:12.585	1.25%	3:15.166	2.61%
12 scen.	3:12.484	1.20%	3:15.017	2.61%
13 scen.	3:13.141	1.54%	3:15.191	2.62%
14 scen.	3:12.737	1.33%	3:15.162	2.61%

Figure 7.1: Job completion times in the Matmul test according to configuration.

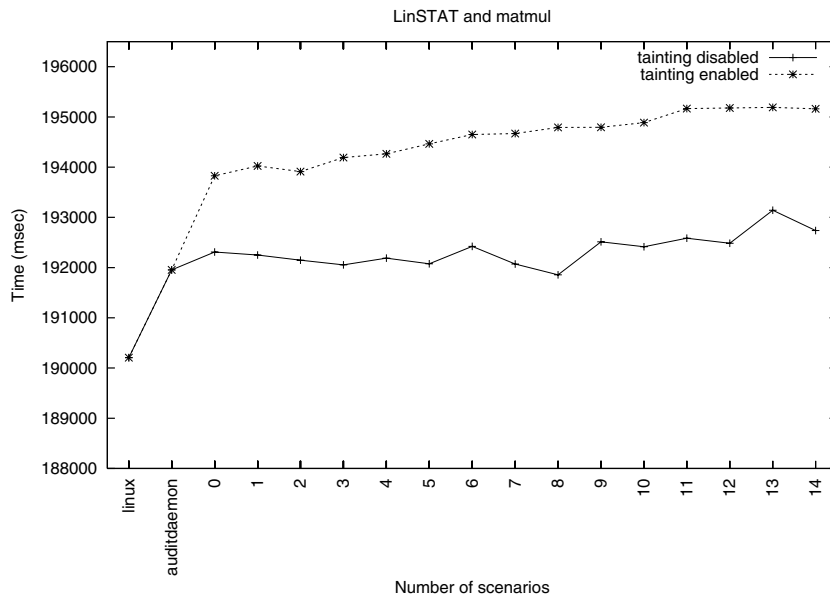


Figure 7.2: Result of Matmul test.

Compile test

<i>Conf</i>	<i>time</i>	<i>Overhead</i>	<i>time taint</i>	<i>Overhead taint</i>
base	7:39.043	-	-	-
auditdaemon	7:48.771	2.12%	-	-
0 scen.	7:51.751	2.77%	7:53.963	3.25%
1 scen.	7:50.848	2.57%	7:51.446	2.70%
2 scen.	7:51.236	2.66%	7:54.983	3.47%
3 scen.	7:51.070	2.62%	7:54.877	3.45%
4 scen.	7:50.900	2.58%	7:56.012	3.70%
5 scen.	7:52.660	2.97%	7:58.556	4.25%
6 scen.	7:52.127	2.85%	7:57.032	3.92%
7 scen.	7:53.246	3.09%	8:00.421	4.66%
8 scen.	7:53.378	3.12%	8:00.117	4.59%
9 scen.	7:53.441	3.14%	8:00.736	4.73%
10 scen.	7:55.350	3.55%	8:01.249	4.84%
11 scen.	7:58.471	4.23%	8:02.212	5.05%
12 scen.	7:59.510	4.46%	8:03.882	5.41%
13 scen.	7:59.032	4.35%	8:03.618	5.35%
14 scen.	7:58.642	4.27%	8:03.423	5.31%

Figure 7.3: Job completion times in the Compile test according to configuration.

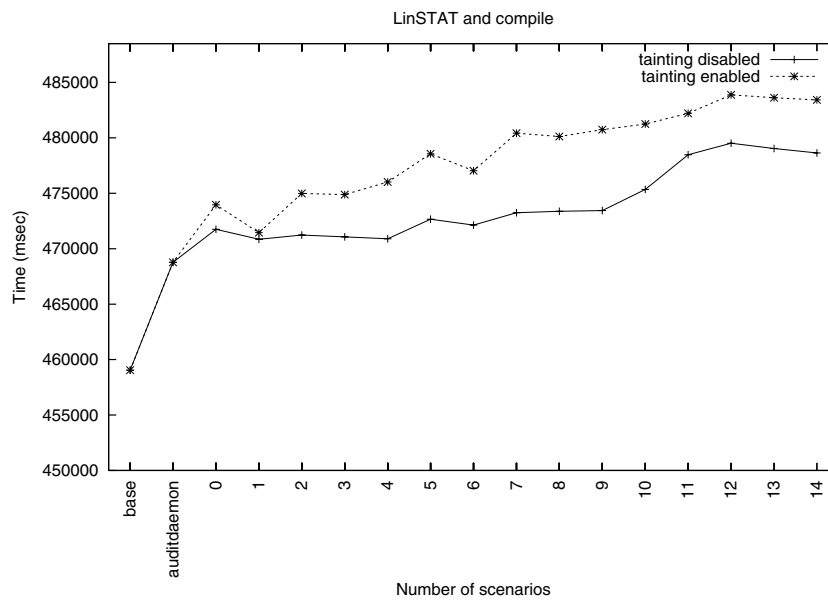


Figure 7.4: Result of Compile test.

non significant increase in the job completion time. Thus, most part of the overhead due to the auditing mechanism is to be attributed to the logging task.

To reduce the impact of logging, it is possible to customize the amount of data that the `auditdaemon` writes at a time. This allows to tune the frequency of writes and, as a consequence, the logging cost.

Passing Data to the Analyzer

The impact of moving data from the kernel to the `auditdaemon` and from the `auditdaemon` to the STAT Core can be evaluated by comparing the configuration with 0 scenarios loaded and the base configuration. As the percentages show, this cost is between 2% and 4%.

Some optimization seems possible to gain better performance in this process. In particular, if the `auditdaemon` used the `mmap` system call to map the kernel audit buffers into its address space, it could avoid one data copy step. Another promising optimization method requires to compress data before transferring it from kernel to user-space. However, it is important to balance the advantage of having a reduced data size with the disadvantage of pre-processing and post-processing data for compression. In the current implementation this feature is not enabled. Nonetheless, data types and structures are optimized for size.

Detecting Intrusions

The impact of processing the audit stream and matching it against attack scenarios can be computed comparing the configurations with at least one scenario with the configuration with 0 scenarios.

As we expected, the overhead increases as the number of scenarios activated grows. This is due to the fact that a larger number of scenario instances are created and need to be examined for matches with the audit input. However, it is difficult to give an exact characterization of this increase, because the processing requirements of a scenario depend on its static and dynamic properties. For example, a scenario with many states and transitions has a bigger impact than one with only a few. Also, if the audit streams is such that a high number of audit events match the active transitions' assertions a larger number of state instances are created and need to be inspected for matches against future events. That said, our set of experiments shows an increase in processing time that is bounded by a linear increase.

The impact of the tainting mechanism can be evaluated by comparing the results obtained in configurations with tainting and configurations without it.

As expected, the tainting mechanism worsens performance: it is a component that requires to maintain state, and, as such, increases the complexity and computational needs of the system. In order to limit the additional overhead, tainting is implemented using data structures with constant lookup and manipulation time, like hash tables. The results are encouraging, though: the tainting mechanism adds only an additional 1% to the total overhead.

Memory and Disk Requirements

During the compile test, the audit trace requires 45MB of disk space, i.e. the trace is created at the speed of about 0.1 MB/sec. However, the audit trace can be compressed on the fly, e.g., using programs like `logrotate`, down to 3.5MB.

The memory usage varied in our tests between a minimum of 12MB and a maximum of 18MB.

7.2 Functional Testing

As we observed in Section 2.4 it is difficult to determine how good is an IDS, and in this case, LinSTAT, at detecting attacks. The most prominent problems are that

1. There are different metrics that we can use to evaluate an IDS and, even more important,
2. For many of these metrics, there are no standardized methods to compute them.

Considering, for instance, the metric “detection performance”, there is no accessible standard data set that can be used to provide a meaningful quantitative result (the data set that was used in the MIT Lincoln Lab test of 1998 only contains network traffic, so it’s clearly not helpful in our case).

The approach we followed for the functional testing of LinSTAT consisted in building a database of exploits and vulnerable programs (similarly to [41]), running the attacks, and checking whether existing scenarios can detect them. Thus, in absence of better comparison terms and metrics, we can say that the functional capabilities of LinSTAT comprises the ability to detect all the attacks that are in our attack database and “similar attacks” (attacks that use the same technique, etc.).

The scenarios presented in the following sections show that LinSTAT can perform policy-based intrusion detection, detect particular classes of buffer

overflow attacks, detect some classes of attacks that modify the flow of control of a program, and can perform detection of specific exploits. These scenarios are useful to understand which parts of the attack space LinSTAT is able, at this moment, to cover.

7.2.1 Scenario `linux_buf_overflow`

This scenario detects a particular class of buffer overflow attacks against SUID programs. In particular, it detects attacks that inject a shellcode in the program through the command line. An EXECUTE event is considered part of an attack if the command line arguments string is longer than `MAX_LEN` (by default, 128 characters) and contains non printable ASCII characters (as returned by `isascii(3)`). Even though this scenario can be bypassed by a determined attacker, e.g., using alphanumeric shellcodes [70], or finding a different way to inject the shellcode in the program, e.g., by setting environment variables, it proved its utility detecting several actual exploits, in particular one taking advantage of a vulnerability in the program `traceroute`¹.

STD Diagram

The State Transition Diagram for this attack is shown in Figure 7.5. It simply comprises of the initial and final state. The state transition is triggered whenever is generated an EXECUTE event with a long and binary `exec_args` parameter.

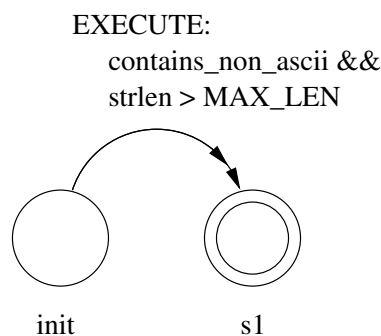


Figure 7.5: STD for the scenario `linux_buf_overflow`.

¹A description of this vulnerability can be found at <http://online.securityfocus.com/bid/1739>.

STATL Scenario

```
use linux;

scenario linux_buf_overflow
{

    // IDMEF Keywords - will be used as part of IDMEF alerts

    string CLASSIFICATION_NAME = "linux_buf_overflow";
    string CLASSIFICATION_URL = "http://www.cs.ucsb.edu/~rsg/"
                                "STAT/linux_buf_overflow.html";

    // Actual User
    string SOURCE_USERNAME = "unknown";
    string SOURCE_USERID = "unknown";
    string SOURCE_USERGROUP = "unknown";

    // Process Keywords
    string SOURCE_PROC_PATH = "unknown";
    string SOURCE_PROC_PID = "unknown";
    string SOURCE_PROC_NAME = "unknown";

    string ADDITIONAL_DATA_DIR = "unknown";
    string ADDITIONAL_DATA_FILE = "unknown";

    // Impact of this attack
    string IMPACT = "attempted-admin";

    int user;
    int pid;

    const int MAX_LEN = 128;

    initial state init {}

    transition exec1 (init -> s1) consuming
    {
        [EXECUTE e1] :
        (permitted(S_ISUID, e1.perm)) &&
    }
}
```

```

(e1.exec_args != NULL) &&
(strlen(e1.exec_args) > MAX_LEN) &&
contains_non_ascii(e1.exec_args) &&
(e1.euid != e1.uid)
{
    user = e1.uid;
    SOURCE_USERID = toString(user);
    SOURCE_USERNAME = userid2name(user, stat);
    SOURCE_USERGROUP = toString(e1.gid);

    pid = e1.pid;
    SOURCE_PROC_PATH = e1.objname;
    SOURCE_PROC_NAME = e1.exec_args;
    SOURCE_PROC_PID = toString(e1.pid);

    ADDITIONAL_DATA_DIR = getDirname(e1.objname);
    ADDITIONAL_DATA_FILE = getFilename(e1.objname);

    if (e1.retcode == 0)
        IMPACT = "successful-admin";
}
}

state s1
{
    {
        IMPACT = "High";
        log("%d (%s): program %s",
            user, SOURCE_USERNAME, SOURCE_PROC_PATH);
    }
}
}

```

Alert

When the scenario detects an occurrence of the modeled attack, an IDMEF alert similar to the following one is generated:

```

<IDMEF-Message version="0.3">
  <Alert ident="shadow-fU5ZzZ*" impact="High">
    <CreateTime ntpstamp="0xC1155636.0x4B41B2F7">

```

```

2002-08-27T01:36:54Z</CreateTime>

<Classification origin="unknown">
  <name>linux_buf_overflow</name>

  <url>http://www.cs.ucsb.edu/~rsg/STAT</url>
</Classification>

<Analyzer analyzerid="LinSTAT:1.0:128.111.48.20">
  <Process>
    <name>LinSTAT:1.0</name>
  </Process>

  <Node>
    <Address category="ipv4-addr">
      <address>128.111.48.20</address>
    </Address>
  </Node>
</Analyzer>

<Source spoofed="unknown">
  <User>
    <UserId type="current-user">
      <name>root</name>

      <number>0</number>
    </UserId>

    <UserId type="current-group">
      <name>0</name>
    </UserId>
  </User>

  <Process>
    <name>./traceroute -g 123 -g 0x9d.0x30.0xfb.0xb7
AAA&#254;&#255;&#255;&#255;&#255;&#255;&#255;&#255;
&#184;&#164;&#255;&#255;&#191;XXX&#235;
V&#205;1&#219;YYZZ&#216;@&#205;&#232;&#220;&#255;
&#255;&#255;/bin/shX</name>

    <path>

```

```

        /home/mcova/bin/traceroute</path>

        <pid>16818</pid>
    </Process>
</Source>

    <AdditionalData type="string"
                                meaning="ustat:target_directory">
/home/mcova/bin</AdditionalData>

    <AdditionalData type="string" meaning="ustat:target_file">
traceroute</AdditionalData>
</Alert>
</IDMEF-Message>

```

The generated alert allows one to identify the offending program and the invocation that caused the alarm to be triggered. Additional information is available to determine the sensor that detected the attack, the node where the attack manifested, and the user responsible for running the attack.

7.2.2 Scenario linux_anomal_execve

This scenario exemplifies how LinSTAT can be used to perform specification-based intrusion detection. This approach uses specifications to describe the intended behavior of a program. The intrusion detection system then detects operations that are in violation of the specifications.

The goal of this scenario is to detect when a program's flow of control is altered by an attacker and induced to execute a different program. This is a typical situation with many exploits for common vulnerabilities, e.g., buffer overflows, format string flaws, or heap corruptions. The attacker is able to make the vulnerable application execute code of his/her choice, which, most often, triggers the execution of a shell. This scenario detects illegal EXECUTE events. The LinSTAT Factbase is used to store which programs are legitimately executed by the monitored processes. Whenever a monitored process executes a program different from those listed in the Factbase, an alert is generated. Note that this approach doesn't give a complete specification of the monitored programs' behavior (as a complete specification-based approach would require), but is sufficient to detect most of the tested exploits.

We validated this scenario against a number of actual attacks, e.g., the

already mentioned `traceroute` exploit, a recent `pine` buffer overflow², and an old `wu-ftpd` exploit³. Note that this scenario only detects attacks that run a new program, typically a shell. It cannot detect attacks that, instead, perform some other malicious action, e.g., adding a new entry in the password file. Moreover, it cannot be used to detect attacks against programs that, as part of their legitimate behavior, execute arbitrary programs, e.g., the SSH daemon.

STD Diagram

The State Transition Diagram for this attack is shown in Figure 7.6. When an EXECUTE event is generated that models the request of execution of a program not contained in the `PROGRAM_RESTRICTED_FILESET` section of the Factbase, the transition from the initial to the final state is triggered.

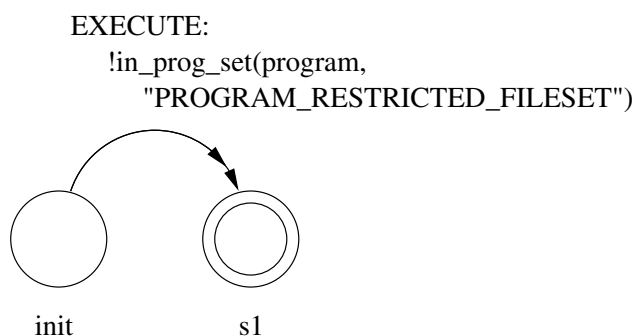


Figure 7.6: STD for the scenario `linux_anomal_execve`.

STATL Scenario

```
use linux;
```

```
scenario linux_anomal_execve
{
  // IDMEF Keywords - will be used as part of IDMEF alerts
  string CLASSIFICATION_NAME = "linux_anomal_execve";
  string CLASSIFICATION_URL =
    "http://www.cs.ucsb.edu/~rsg/STAT/linux_anomal_execve";
}
```

²The attack details are available at <http://www.iddefense.com/advisory/09.10.03.txt>

³The exploit is available at <http://www.project-hack.org/exploits/new/0x82-wu262.c>

```

// Actual User
string SOURCE_USERNAME = "unknown";
string SOURCE_USERID = "unknown";
string SOURCE_USERGROUP = "unknown";

// Process Keywords
string SOURCE_PROC_PATH = "unknown";
string SOURCE_PROC_PID = "unknown";
string SOURCE_PROC_NAME = "unknown";

string TARGET_PROC_PATH = "unknown";

string ADDITIONAL_DATA_DIR = "unknown";
string ADDITIONAL_DATA_FILE = "unknown";

// Impact of this attack
string IMPACT = "attempted-admin";

int user;
int pid;

string monitored_program = "traceroute";

initial state init {}

transition anomal_execute (init -> s1) consuming
{
    [EXECUTE e1] :
        (
            strstr(e1.pathname, monitored_program.c_str()) &&
            !in_prog_set(e1.objname, "TEST_RESTRICT_FILESET", stat)
        ) ||
        // Apache monitoring.
        // Other programs to be monitored can be added similarly
        (
            !strcmp(e1.pathname, "/usr/sbin/httpd") &&
            !in_prog_set(e1.objname, "APACHE_RESTRICT_FILESET",
                stat)
        )
}

```

```

) ||
// Programs in NO_EXECVE_FILESET do not call execve(2)
in_prog_set(e1.pathname, "NO_EXECVE_FILESET", stat)
{
    user = e1.uid;
    SOURCE_USERID = toString(user);
    SOURCE_USERNAME = userid2name(user, stat);
    SOURCE_USERGROUP = toString(e1.gid);

    pid = e1.pid;
    SOURCE_PROC_PATH = e1.pathname;
    SOURCE_PROC_NAME = e1.exec_args;
    SOURCE_PROC_PID = toString(e1.pid);

    TARGET_PROC_PATH = e1.objname;

    ADDITIONAL_DATA_DIR = getDirname(e1.objname);
    ADDITIONAL_DATA_FILE = getFilename(e1.objname);

    if (e1.retcode == 0)
        IMPACT = "successful-admin";
}
}

state s1
{
    {
        log("%d (%s): %s executed %s", user,
            SOURCE_USERNAME.c_str(), SOURCE_PROC_NAME.c_str(),
            TARGET_PROC_PATH.c_str());
    }
}
}

```

Alert Excerpt

The following alert excerpt allows one to determine the user responsible of the attack. Additional information, not shown here, is available to identify the sensor that detected the attack, the node where the attack manifested, and the attacked program.

```

<User>
  <UserId type="current-user">
    <name>mcova</name>
    <number>502</number>
  </UserId>
  <UserId type="current-group">
    <name>502</name>
  </UserId>
</User>

```

7.2.3 Scenario `linux_system_program_write`

This scenario detects the implantation of trojan horses or backdoored copies of system executables. The Factbase is used to list the system directories that should be monitored and the programs that are allowed to write in these directories. All other write accesses to these directories are flagged as malicious.

In our tests, no program was allowed to write in directories containing system programs and configuration files, i.e., `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, and `/etc`. Besides correctly detecting our test cases, this scenario helped finding a misconfiguration of the web server that tried to write log information in the `/etc` directory.

STD Diagram

Figure 7.7 shows the STD diagram for this scenario. It has two states: the initial and final one. The state transition is triggered in correspondence of a WRITE event for an executable file located in a system directory.

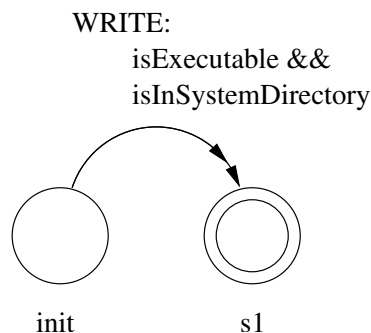


Figure 7.7: STD for the scenario `linux_system_program_write`.

STATL Scenario

```
use linux;

scenario linux_system_program_write
{
    // IDMEF Keywords - will be used as part of IDMEF alerts

    string CLASSIFICATION_NAME = "system_program_write";
    string CLASSIFICATION_URL = "http://www.cs.ucsb.edu/~rsg/"
        "STAT/linux_system_program_write";

    // Actual User
    string SOURCE_USERNAME = "unknown";
    string SOURCE_USERID = "unknown";
    string SOURCE_USERGROUP = "unknown";

    // Process Keywords
    string SOURCE_PROC_PATH = "unknown";
    string SOURCE_PROC_PID = "unknown";
    string SOURCE_PROC_NAME = "unknown";

    // Target
    string TARGET_PROC_PATH = "unknown";

    string IMPACT = "attempted-admin";

    string ADDITIONAL_DATA_DIR = "unknown";
    string ADDITIONAL_DATA_FILE = "unknown";

    int pid;
    int auid;

    initial
    state init { }

    transition write (init->s1) nonconsuming
    {
        [WRITE w] :
            !ignore_this(w.objname, WRITE_ID, stat) &&
            permitted(S_IXOTH, w.perm) &&
    }
}
```

```

        member(w.objname,
            "RESTRICTED_WRITE_EXECUTABLE_FILES", stat) &&
        !in_user_set(w.uid,
            "RESTRICTED_WRITE_EXECUTABLE_FILES", stat) &&
        !in_group_set(w.gid,
            "RESTRICTED_WRITE_EXECUTABLE_FILES", stat) &&
        !in_prog_set(w.pathname,
            "RESTRICTED_WRITE_EXECUTABLE_FILES", stat)
    {
        auid = w.uid;
        SOURCE_USERNAME = userid2name(auid, stat);
        SOURCE_USERID = toString(auid);
        SOURCE_USERGROUP = toString(w.gid);
        pid = w.pid;
        SOURCE_PROC_PATH = w.pathname;
        SOURCE_PROC_NAME = w.exec_args;
        SOURCE_PROC_PID = toString(pid);

        TARGET_PROC_PATH = w.objname;
        ADDITIONAL_DATA_DIR = getDirname(TARGET_PROC_PATH);
        ADDITIONAL_DATA_FILE = getFilename(TARGET_PROC_PATH);

        if (w.retcode >= 0)
            IMPACT = "successful-admin";
    }
}

state s1
{
    {
        log("%d: %s written by user %d, process %d",
            auid, SOURCE_PROC_PATH.c_str(), auid, pid);
    }
}
}

```

Alert Excerpt

The following alert excerpt shows that somebody tried to overwrite the program `/bin/echo`. The full alert reports other information allowing to easily identify the user responsible of the attack, the sensor that detected the at-

tack, and the node where the attack happened.

```
<Target>
  <Process>
    <name>/bin/echo</name>
    <path>/bin/echo</path>
  </Process>
</Target>
```

7.2.4 Scenario `restricted_read`

This scenario shows how LinSTAT can be used to detect policy infringement. The LinSTAT Factbase is used to specify that certain files should be accessed only by certain users and by using specific programs. This scenario raises an alarm whenever the policy specified in the Factbase is violated. Other scenarios have been similarly designed to detect infringement of policies regulating writing of files and access to directories.

STD Diagram

The STD diagram for this scenario is shown in Figure 7.8. An event modeling a READ attempt of a restricted object by an unauthorized program triggers the transition from the initial to the final state.

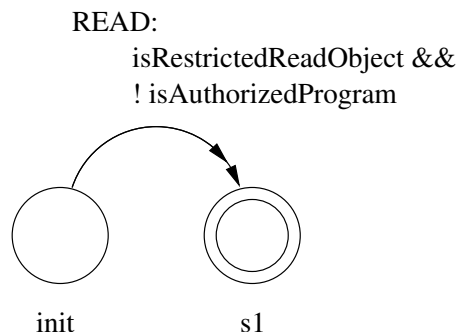


Figure 7.8: STD for the scenario `linux_restricted_read`.

STATL Scenario

```
use linux;

scenario linux_restricted_read
{
```

```

// IDMEF Keywords - will be used as part of IDMEF alerts

string CLASSIFICATION_NAME = "restricted_read";
string CLASSIFICATION_URL =
    "http://www.cs.ucsb.edu/~rsg/STAT/linux_restricted_read";
string IMPACT = "unknown";

// Actual User
string SOURCE_USERNAME = "unknown";
string SOURCE_USERID = "unknown";
string SOURCE_USERGROUP = "unknown";

// Process Keywords
string SOURCE_PROC_PATH = "unknown";
string SOURCE_PROC_PID = "unknown";
string SOURCE_PROC_NAME = "unknown";

// Target info
string TARGET_PROC_PATH = "unknown";

string ADDITIONAL_DATA_FILE = "unknown";
string ADDITIONAL_DATA_DIR = "unknown";

int auid;
int pid;

initial
state init { }

transition read (init->s1) consuming
{
    [READ r] :
        r.retcode >= 0 &&
        member(r.objname, "RESTRICTED_READ_OBJECTS", stat) &&
        !in_prog_set(r.pathname,
                    "RESTRICTED_READ_OBJECTS", stat) &&
        !ignore_this(r.objname, r.act, stat)
        {
            SOURCE_USERNAME = userid2name(r.uid, stat);
            SOURCE_USERID = toString(r.uid);
        }
}

```

```

SOURCE_USERGROUP = toString(r.gid);

SOURCE_PROC_PATH = r.pathname;
SOURCE_PROC_PID = toString(r.pid);
SOURCE_PROC_NAME = r.exec_args;

TARGET_PROC_PATH = r.objname;

IMPACT = "bad-unknown";

    auid = r.auid;
    pid = r.pid;
}
}

state s1
{
    {
        ADDITIONAL_DATA_DIR = getDirname(TARGET_PROC_PATH);
        ADDITIONAL_DATA_FILE = getFilename(TARGET_PROC_PATH);

        log("%d: %s read by user %d, program %s",
            auid, TARGET_PROC_PATH.c_str(), auid,
            SOURCE_PROC_PATH.c_str());
    }
}
}

```

Alert Excerpt

The following alert excerpt shows that the file `/proc/devices` was accessed infringing the site policy specified in the LinSTAT Factbase. The complete alert reports information about the user that infringed the policy and the node where this happened.

```

<Target>
  <Process>
    <name>/proc/devices</name>
    <path>/proc/devices</path>
  </Process>
</Target>

```

7.2.5 Scenario `linux_traverse_vulnerability`

This scenario detects exploitations of tree traversal vulnerabilities. Some programs, typically web servers, provide access to files contained in a restricted local directory tree to remote users. Files outside of this intended accessible tree should not be accessed by remote users through these programs. However, if such a program contains a tree traversal vulnerability, it is possible for an attacker to craft a special request that tricks the vulnerable program into access files outside the intended directory tree. Directories that should be accessed by a program can be stored in the Factbase. We used this scenario to detect attacks that exploited a tree traversal vulnerability in **Interchange**, a tool to develop e-commerce web sites⁴. In particular, we extended the Factbase with the `INTERCHANGE_RESTRICT_FILESET` section that lists the filesystem's subtrees that can be read by the Interchange program.

STD Diagram

Figure 7.9 shows the STD diagram for this scenario. If the Interchange process attempts to read a file outside the directory subtree specified in the `INTERCHANGE_RESTRICT_FILESET`, a `READ` event is generated that triggers the transition from the initial to the final state.

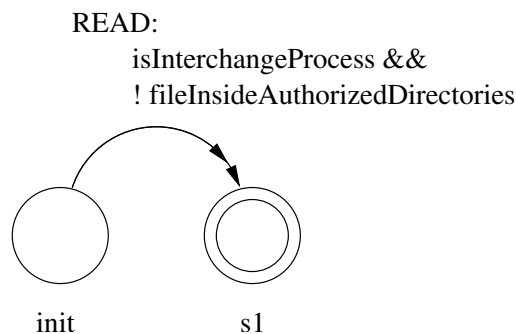


Figure 7.9: STD for the scenario `linux_traverse_vulnerability`.

STATL Scenario

```
use linux;  
  
scenario linux_traverse_vulnerability  
{
```

⁴This vulnerability is described at <http://online.securityfocus.com/bid/5453>.

```

// IDMEF Keywords - will be used as part of IDMEF alerts
// generated by ustat

string CLASSIFICATION_NAME = "linux_traverse_vulnerability";
string CLASSIFICATION_URL =
    "http://online.securityfocus.com/bid/5453";

// Actual User
string SOURCE_USERNAME = "unknown";
string SOURCE_USERID = "unknown";
string SOURCE_USERGROUP = "unknown";

// Process Keywords
string SOURCE_PROC_PATH = "unknown";
string SOURCE_PROC_PID = "unknown";
string SOURCE_PROC_NAME = "unknown";

string TARGET_PROC_PATH = "unknown";

string ADDITIONAL_DATA_DIR = "unknown";
string ADDITIONAL_DATA_FILE = "unknown";

// Impact of this attack
string IMPACT = "attempted-admin";

int user;
int pid;

const int MAX_LEN = 128;

initial state init {}

transition read_traverse (init -> s1) consuming
{
    [READ e1] :
        !strcmp(e1.exec_args, "interchange") &&
        !in_prog_set(e1.objname,
            "INTERCHANGE_RESTRICT_FILESET", stat) &&
        !member_dir_sub(e1.objname,
            "INTERCHANGE_RESTRICT_FILESET", stat)

```

```

    {
        user = e1.uid;
        SOURCE_USERID = toString(user);
        SOURCE_USERNAME = userid2name(user, stat);
        SOURCE_USERGROUP = toString(e1.gid);

        pid = e1.pid;
        SOURCE_PROC_PATH = e1.pathname;
        SOURCE_PROC_NAME = e1.exec_args;
        SOURCE_PROC_PID = toString(e1.pid);
        TARGET_PROC_PATH = e1.objname;

        ADDITIONAL_DATA_DIR = getDirname(e1.objname);
        ADDITIONAL_DATA_FILE = getFilename(e1.objname);

        if (e1.retcode >= 0)
            IMPACT = "successful-admin";

        log("%d (%s): %s read %s outside of its tree during "
            "connection from %s:%hd",
            user, SOURCE_USERNAME.c_str(), e1.exec_args,
            e1.objname, getSrcAddrStr(e1.pid, stat).c_str(),
            getSrcPort(e1.pid, stat));
    }
}

state s1
{
    {
        log("%d (%s): %s traverse vulnerability bug "
            "exploited: %s accessed", user,
            SOURCE_USERNAME.c_str(), SOURCE_PROC_NAME.c_str(),
            TARGET_PROC_PATH.c_str());
    }
}
}

```

Alert Excerpt

The following alert excerpt allows one to determine what files were illegitimately accessed, in this case the password file. The complete alert reports

other information allowing to easily identify the user responsible of the attack, the sensor that detected the attack, and the node where the attack happened.

```
<Target>  
  <path>/etc/passwd</path>  
</Target>
```

7.2.6 Scenario qpopper

This scenario models an attack against `qpopper`⁵. The attack takes advantage of a local arbitrary command execution vulnerability in `qpopper`. It creates two C programs, compile them obtaining the programs `/tmp/x82` and `/tmp/x0x` and remove the sources. It then forks and exec the `poppassd` program with the flag `-s` and the corresponding parameter set to `/tmp/x0x`. Finally `/tmp/x82` is executed. This scenario models the attack as follows: the attacker creates a first program and a second program. Then `poppassd`, the second program (`/tmp/x0x`), and the first program (`/tmp/x82`) are executed in this order.

Note that, while this scenario achieves good results in terms of generation of false positives (because it models in detail the evolving of this specific attack), an attacker could create a modified version of the exploit that doesn't trigger an alert. This is a typical problem of attack modeling: the more detailed is the attack model, the less false positives are generated, but, at the same time, the easier for an attacker to create a variation of the attack that is not covered by the model and thus remains undetected. Clearly, a trade-off between false positives generation and resilience to attack modification is needed.

This scenario is also a good example of the level of sophistication that can be reached in modeling attacks using the STATL language.

STD Diagram

The STD for this scenario is shown in Figure 7.10. The attack proceeds as described above. All the forward transitions but the last one are nonconsuming: occurrences of the attack can take place starting from every intermediate state. Note the unwinding transitions that bring back the system to the initial state if the first or the second program are deleted before their use or if the `poppassd` program exits before it executes the second program.

⁵The attack is described at <http://www.securityfocus.com/bid/7447>.

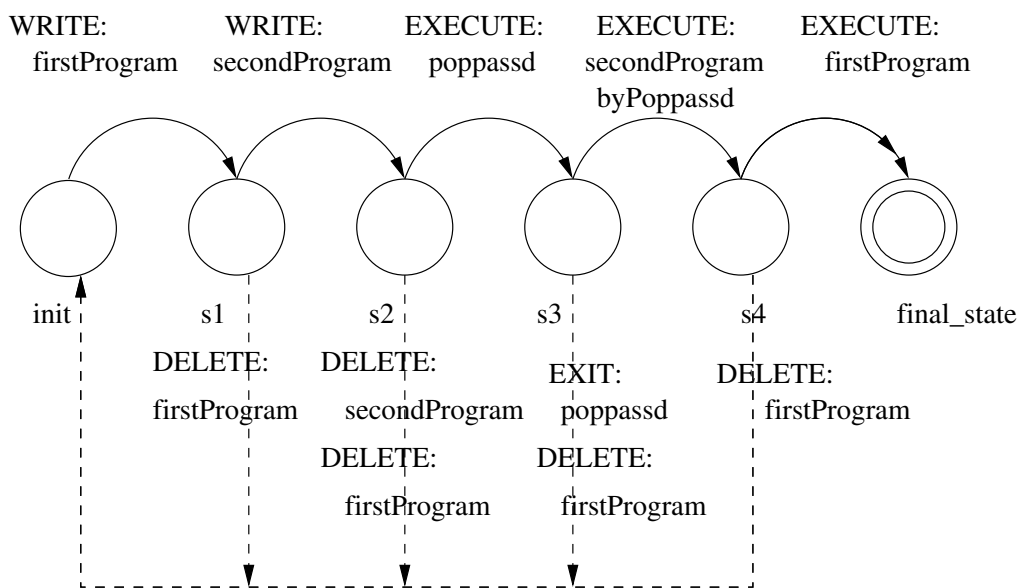


Figure 7.10: STD for the scenario linux_qpopper.

STATL Scenario

```
use linux;
```

```
scenario linux_qpopper
```

```
{
```

```
  // IDMEF Keywords - will be used as part of IDMEF alerts
  string CLASSIFICATION_NAME = "linux_qpopper";
  string CLASSIFICATION_URL =
    "http://www.securityfocus.com/bid/7447";
```

```
  // Actual User
```

```
  string SOURCE_USERNAME = "unknown";
  string SOURCE_USERID = "unknown";
  string SOURCE_USERGROUP = "unknown";
```

```
  // Process Keywords
```

```
  string SOURCE_PROC_PATH = "unknown";
  string SOURCE_PROC_PID = "unknown";
  string SOURCE_PROC_NAME = "unknown";
```

```
  string ADDITIONAL_DATA_DIR = "unknown";
```

```
  string ADDITIONAL_DATA_FILE = "unknown";
```

```

string TARGET_PROC_PID = "unknown";
string TARGET_PROC_PATH = "unknown";
string TARGET_PROC_NAME = "unknown";
string TARGET_USERNAME = "unknown";
string TARGET_USERID = "unknown";
string TARGET_NODEADDRESS = stat_hostname_get(stat);

// Impact of this attack
string IMPACT = "successful-admin";

// pid of the offending process
int pid;
// uid of the attacker
int userid;
// inodes of the two programs created for the attack
long first_program_inode;
long second_program_inode;
// pid of poppassd
int poppassd_pid;
// pathname of poppassd
string poppassd_pathname = "/usr/local/bin/poppassd";

initial state init {}

transition create_first_program (init -> s1)
  nonconsuming
{
  [WRITE c1] :
    c1.retcode >=0
    {
      pid = c1.pid;
      first_program_inode = c1.inode;
    }
}

state s1 { }

transition create_second_program (s1 -> s2) nonconsuming
{

```

```

[WRITE c1] :
    c1.retcode >= 0
    {
        second_program_inode = c1.inode;
        pid = c1.pid;
    }
}

state s2 { }

transition execute_poppassd (s2 -> s3) nonconsuming
{
    [EXECUTE e1] :
        !strcmp(e1.objname, poppassd_pathname.c_str()) &&
        e1.retcode >= 0
        {
            userid = e1.uid;
            poppassd_pid = e1.pid;
        }
}

state s3 { }

transition executed_second_program (s3 -> s4) consuming
{
    [EXECUTE e1] :
        e1.inode == second_program_inode &&
        e1.ppid == poppassd_pid &&
        e1.retcode >= 0
        { }
}

state s4
{
    {
        log("/tmp/x0x (inode: %ld) executed via poppassd\n",
            second_program_inode);
    }
}

transition executed_first_program (s4 -> final_state)

```

```

consuming
{
  [EXECUTE e1] :
    e1.retcode >= 0 &&
    e1.inode == first_program_inode
    {
      SOURCE_USERNAME = userid2name(e1.uid, stat);
      SOURCE_USERID = toString(e1.uid);
      SOURCE_USERGROUP = toString(e1.gid);
      TARGET_PROC_PID = toString(e1.pid);
      TARGET_PROC_PATH = e1.objname;
      TARGET_PROC_NAME = e1.exec_args;
      TARGET_USERNAME = userid2name(e1.owner, stat);
      TARGET_USERID = toString(e1.owner);
    }
}

state final_state
{
  {
    log("User %d exploited qpopper/poppassd flaw.\n",
        userid);
  }
}

transition unwind1 (s1 -> init) unwinding
{
  [DELETE e] : e.inode == first_program_inode
  { }
}

transition unwind2 (s2 -> init) unwinding
{
  [DELETE e] : e.inode == second_program_inode
  { }
}

transition unwind3(s3 -> init) unwinding
{
  [EXIT e] : e.pid == poppassd_pid
  { }
}

```

```
}  
}
```

Alert Excerpt

The following alert excerpt identifies the attacker that exploited the qpopper flaw. The complete alert reports other information allowing to determine the sensor that detected the attack, and the node where the attack happened.

```
<User>  
  <UserId type="current-user">  
    <name>mcova</name>  
    <number>502</number>  
  </UserId>  
  <UserId type="current-group">  
    <name>502</name>  
  </UserId>  
</User>
```

7.2.7 Scenario openssl

This scenario models an attack against Apache/mod_ssl servers⁶.

The attack exploits a vulnerability in the OpenSSL code to inject a shellcode. The injected shellcode is two-staged. In the first stage, the shellcode executes a total of 32 forks, reads, writes and reads. In the second stage, it resets its uid, euid and saved uid to 0 and then executes /bin/sh. In this scenario, we just model the second-stage shellcode.

This scenario uses the tainting mechanism: it only monitors instances of the Apache program generated in consequence of a network connection to the local machine.

STD Diagram

The STD for this scenario is shown in Figure 7.11. It has three states: `init`, `s1` and `final_state`. The transition from `init` to `s1` is triggered when a tainted instance of the Apache program tries to set its real and effective user ID to zero. The transition from `s1` to the final state happens if the same process executes a shell. Unwinding transitions handle the case of the tainting process finishing or resetting its identity to an ID different from zero.

⁶The attack is available online at <http://packetstormsecurity.org/0209-exploits/openssl-too-open.tar.gz>

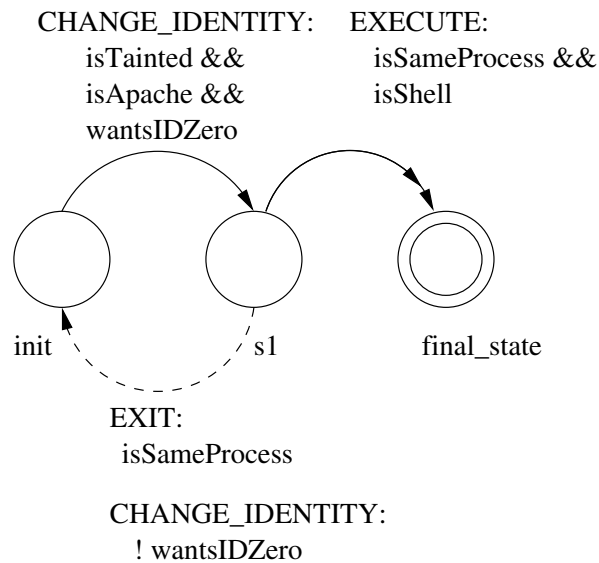


Figure 7.11: STD for the scenario linux_openssl.

STATL Scenario

```

use linux;

scenario linux_openssl
{

    // IDMEF Keywords - will be used as part of IDMEF alerts
    string CLASSIFICATION_NAME = "linux_openssl_overflow";
    string CLASSIFICATION_URL =
        "http://packetstormsecurity.org/0209-exploits/"
        "openssl-too-open.tar.gz";

    // Actual User
    string SOURCE_USERNAME = "unknown";
    string SOURCE_USERID = "unknown";
    string SOURCE_USERGROUP = "unknown";

    // Process Keywords
    string SOURCE_PROC_PATH = "unknown";
    string SOURCE_PROC_PID = "unknown";
    string SOURCE_PROC_NAME = "unknown";

```

```

string TARGET_PROC_PID = "unknown";
string TARGET_PROC_PATH = "unknown";
string TARGET_PROC_NAME = "unknown";
string TARGET_USERNAME = "unknown";
string TARGET_USERID = "unknown";

string SOURCE_NODEADDRESS= "unknown";
string TARGET_NODEADDRESS = stat_hostname_get(stat);

// Impact of this attack
string IMPACT = "unknown";

// where the attacker is coming from,
// if the source IP is not spoofed
string origin_ip = "unknown";

// pid of the offending process
int pid;

// full pathname of the httpd executable
string httpd_pathname = "/usr/local/apache/bin/httpd";

initial state init {}

transition change_id (init -> s1) nonconsuming
{
    [CHANGE_IDENTITY c1] :
        isTainted(c1.pid, stat) &&
        c1.euid == 0 &&
        c1.uid == 0 &&
        !strcmp(httpd_pathname.c_str(), c1.pathname)
        {
            origin_ip = getSrcAddrStr(c1.pid, stat);
            pid = c1.pid;
        }
}

state s1 { }

transition exec_shell (s1 -> final_state) consuming
{

```

```

[EXECUTE e1]:
(match_fullname(e1.objname, "/bin/bash") ||
match_fullname(e1.objname, "/bin/sh")) &&
e1.retcode >= 0 &&
e1.pid == pid
{
    SOURCE_USERNAME = userid2name(e1.uid, stat);
    SOURCE_USERID = toString(e1.uid);
    SOURCE_USERGROUP = toString(e1.gid);
    TARGET_PROC_PID = toString(e1.pid);
    TARGET_PROC_PATH = e1.objname;
    TARGET_PROC_NAME = e1.exec_args;
    TARGET_USERNAME = userid2name(e1.owner, stat);
    TARGET_USERID = toString(e1.owner);
    SOURCE_NODEADDRESS = origin_ip;
}
}

state final_state
{
    {
        IMPACT = "successful-admin";
        log("Attacker apparently from %s exploited OpenSSL flaw",
            origin_ip.c_str());
    }
}

transition unwind1 (s1 -> init) unwinding
{
    [EXIT e] : e.pid == pid
    { }
}

transition unwind2 (s1 -> init) unwinding
{
    [CHANGE_IDENTITY c2]:
    c2.pid == pid &&
    c2.euid != 0 &&
    c2.uid != 0
    { }
}

```

}

Alert Excerpt

The following alert excerpt allows one to determine the IP address of the machine from which the attack was launched. This, of course, assumes that the attacker did not spoof its IP address. The complete alert reports other information allowing to identify the process that was attacked, the sensor that detected the attack, and the node where the attack happened.

```
<Source spoofed="unknown">
  <Node>
    <Address category="ipv4-addr">
      <address>128.111.48.117</address>
    </Address>
  </Node>
</Source>
```

Capitolo 8

Conclusion and Future Work

In the first part of this thesis we presented Intrusion Detection Systems (IDSs) and discussed their use and application as security tools. We analyzed a taxonomy of IDSs and examined different possible choices in the design space of these systems. We also showed the issues of evaluating and testing IDSs and the open problems that are waiting to be addressed and solved by future research in this field.

In the second part of the thesis we presented the result of our research work: a host-based Intrusion Detection System for the Linux platform named LinSTAT. LinSTAT extends the STAT Framework, a framework for the development of IDSs that provides a number of domain-independent components useful for intrusion detection purposes. In particular, LinSTAT comprises a mechanism to instrument the Linux kernel in order to generate complete and meaningful auditing information and introduces a series of modules that realize platform-specific functionalities. It employs the tainting mechanism, a novel method to track and monitor local activities caused by remote users. In addition, a number of attack scenarios, that model different types of attacks and intrusions, were written and tested against both test cases and actual exploits. The overhead of the auditing system was evaluated and showed a low impact on the performance of the monitored system that makes LinSTAT a viable tool to monitor and protect real-world systems.

We plan to continue our work extending the original project in different directions. As a start, we intend to write more scenarios, possibly combining events originated by different sensors, e.g., a network-based sensor's events coupled with LinSTAT's events. We are also interested in researching a more general and flexible audit mechanism that allows to dynamically specify what events are to be audited and what information is to be collected. We also intend to expand the original LinSTAT sensor in order to perform specification/policy-based host management at the system call level.

The complete source code and documentation of the STAT Framework and LinSTAT is available for download, modification and testing at the address <http://www.cs.ucsb.edu/~rsg/STAT/software/>.

Bibliografia

- [1] M. Abadi and S. Bellovin, editors. *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2002.
- [2] D. Akers, editor. *Proceedings of the 14th Annual Computer Security Applications Conference*. IEEE Computer Society, December 1998.
- [3] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the Practice of Intrusion Detection Technologies. Technical Report CMU/SEI-99TR-028, Carnegie Mellon, Software Engineering Institute, January 2000.
- [4] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. In Lee et al. [49], pages 22–36.
- [5] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, April 1980.
- [6] S. Axelsson. Research in Intrusion-Detection systems: A Survey. Technical Report 98–17, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, December 1998.
- [7] S. Axelsson. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In *ACM Conference on Computer and Communications Security*, pages 1–7, 1999.
- [8] R. Bace and P. Mell. Intrusion Detection Systems. Technical Report NIST SP 800-31, National Institute of Standards and Technology, November 2001.
- [9] S. M. Bellovin. Packets Found on an Internet. *Computer Communications Review*, 23(3):26–31, July 1993.
- [10] S. M. Bellovin. Computer Security - An End State? *Communications of the ACM*, 44(3):131–132, 2001.

- [11] bugtraq. <http://www.securityfocus.com>.
- [12] CERT Coordination Center (CERT/CC). CERT/CC Statistics 1988-2003. http://www.cert.org/stats/cert_stats.html.
- [13] M. J. Crosbie and B. A. Kuperman. A Building Block Approach to Intrusion Detection. http://www.raid-symposium.org/Raid2001/papers/crosbie_kuperman_raid2001.pdf, October 2001.
- [14] F. Cuppens. Managing Alerts in a Multi-Intrusion Detection Environment. In *Proceedings of the 17th Annual Computer Security Application Conference*, December 2001.
- [15] F. Cuppens and A. Mieke. Alert Correlation in a Cooperative Intrusion Detection Framework. In Abadi and Bellovin [1], pages 202–215.
- [16] M. Dacier, H. Debar, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, April 1999.
- [17] H. Debar, M. Dacier, and A. Wespi. A Revised Taxonomy for Intrusion-Detection Systems. Technical report, IBM Research, Zurich Research Laboratory, 1999.
- [18] H. Debar and A. Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In Lee et al. [49], pages 85–103.
- [19] D. E. Denning. An Intrusion-Detection Model. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, Oakland, California, USA, April 1986. IEEE Computer Society Press.
- [20] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Addendum to “Testing and Evaluating Computer Intrusion Detection Systems”. *Communications of the ACM*, 42(9):15, 1999.
- [21] R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Testing and Evaluating Computer Intrusion Detection Systems. *Communications of the ACM*, 42(7):53–61, 1999.
- [22] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. In *Proceedings of the ACM Workshop on Intrusion Detection Systems*, Athens, Greece, November 2000.
- [23] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL Definition. Technical report, UCSB, June 2001.

- [24] C. Faloutsos and S. Christodoulakis. Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation. *ACM Transactions on Information Systems (TOIS)*, 2(4):267–288, 1984.
- [25] R. Feiertag, C. Kahn, P. Porras, D. Schnackenberg, S. Staniford-Chen, and B. Tung. A Common Intrusion Specification Language (CISL). <http://gost.isi.edu/cidf/drafts/language.txt>.
- [26] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, August 2001.
- [27] A. K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In Akers [2], pages 259–267.
- [28] Intrusion Detection Working Group. Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition. <http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-10.txt>.
- [29] J. Haines, D. K. Ryder, L. Tinnel, and S. Taylor. Validation of Sensor Alert Correlators. *IEEE Security & Privacy*, 1(1):46–56, 2003.
- [30] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [31] S. E. Hansen and E. T. Atkins. Automated System Monitoring and Notification With Swatch. In *Proceedings of the Seventh Systems Administration Conference (LISA '93)*, pages 145–151, Monterey, California, 1993.
- [32] P. Helman and G. Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, September 1993.
- [33] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [34] HostSentry. <http://www.psionic.com/products/host Sentry.html>.
- [35] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. Master’s thesis, Computer Science Department, University of California, Santa Barbara, July 1992.

- [36] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 1993.
- [37] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3), March 1995.
- [38] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.
- [39] C. Kahn, P. A. Porras, S. Staniford-Chen, and B. Tung. A Common Intrusion Detection Framework. <http://gost.isi.edu/cidf/papers/cidf-jcs.ps>.
- [40] R. A. Kemmerer and G. Vigna. Intrusion Detection: A Brief History and Overview. *IEEE Computer*, 35(4):27–30, April 2002.
- [41] K. Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. Master’s thesis, Massachusetts Institute of Technology, June 1999.
- [42] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.
- [43] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.
- [44] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In Abadi and Bellovin [1], pages 285–294.
- [45] S. Kumar and E. H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.
- [46] B. A. Kuperman and E. Spafford. Generation of Application Level Data via Library Interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, West Lafayette, Indiana 47907-1398, October 1999.

- [47] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.
- [48] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
- [49] W. Lee, L. Mé, and A. Wespi, editors. *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001, Proceedings*, volume 2212 of *Lecture Notes in Computer Science*. Springer, 2001.
- [50] U. Lindqvist and P. A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999. IEEE Computer Society Press, Los Alamitos, California.
- [51] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyszogrod, R. Cunningham, and M. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Los Alamitos, CA, 2000. IEEE Computer Society Press.
- [52] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In H. Debar, L. Mé, and S. F. Wu, editors, *RAID*, volume 1907 of *Lecture Notes in Computer Science*, pages 162–182. Springer, 2000.
- [53] LogSentry. <http://www.psionic.com/products/logsentry.html>.
- [54] C. Lonvick. The BSD syslog Protocol. RFC 3164, August 2001.
- [55] T. F. Lunt. Detecting Intruders in Computer Systems. In *Proceedings of the Fifth Canadian Conference on Auditing and Computer Technology*, 1993.
- [56] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and Application Protocol Scrubbing. In *INFOCOM (3)*, pages 1381–1390, 2000.
- [57] J. McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed

by Lincoln Laboratory. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):262–294, 2000.

- [58] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman. An Overview of Issues in Testing Intrusion Detection Systems. Technical Report NIST IR 7007, National Institute of Standards and Technology, June 2003.
- [59] D. Mutz, G. Vigna, and R. A. Kemmerer. An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In *Proceedings of the 2003 Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2003.
- [60] P. G. Neumann and P. A. Porras. Experience with EMERALD to Date. In *First USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, April 1999.
- [61] U.S. Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria, December 1985.
- [62] S. Patton, W. Yurcik, and D. Doss. An Achilles Heel in Signature-Based IDS: Squealing False Positives in SNORT. <http://www.raid-symposium.org/raid2001/program.html>.
- [63] P. Porras, D. Schnackenberg, S. Staniford-Chen, M. Stillman, and F. Wu. The Common Intrusion Detection Framework Architecture. <http://www.isi.edu/gost/cidf/drafts/architecture.txt>.
- [64] P. A. Porras and A. Valdes. Live Traffic Analysis of TCP/IP Gateways. In *Internet Society's Networks and Distributed Systems Security Symposium*, March 1998.
- [65] Prelude Hybrid IDS. <http://www.prelude-ids.org/>.
- [66] K. E. Price. Host-Based Misuse Detection and Conventional Operating Systems' Audit Data Collection. Master's thesis, Purdue University, December 1997.
- [67] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.
- [68] N. J. Puketza, M. Chung, R. A. Olsson, and B. Mukherjee. A Software Platform for Testing Intrusion Detection Systems. *IEEE Software*, 14(5):43–51, September/October 1997.

- [69] E. Rescorla. Security holes... Who cares? In V. Paxson, editor, *USENIX Security Symposium*, pages 75–90. USENIX, 2003.
- [70] rix@hert.org. Writing ia32 alphanumeric shellcodes. *Phrack*, 0x0b(0x39), August 2001.
- [71] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, pages 229–238. USENIX, 1999.
- [72] rpm. <http://www.redhat.com/docs/books/max-rpm>.
- [73] SNARE. <http://www.intersectalliance.com/projects/Snare>.
- [74] Snot: An arbitrary packet generator that uses Snort rules files as its source of packet information. <http://www.stolenshoes.net/sniph/>.
- [75] E. Spafford and D. Zamboni. Data Collection Mechanisms for Intrusion Detection Systems. Technical Report CERIAS TR 2000-08, Center for Education and Research in Information Assurance and Security, June 2000.
- [76] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.
- [77] Stick: an IDS stress tool. <http://www.eurocompton.net/stick/projects8.html>.
- [78] Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module*. 2550 Garcia Ave., Mountain View, CA 94043, December 1991.
- [79] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In A. Wespi, G. Vigna, and L. Deri, editors, *RAID*, volume 2516 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2002.
- [80] K. M. C. Tan and R. A. Maxion. Why 6? Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In Abadi and Bellovin [1], pages 188–201.
- [81] A. Valdes and K. Skinner. Probabilistic Alert Correlation. In Lee et al. [49], pages 54–68.
- [82] G. Vigna, S. T. Eckmann, and R. A. Kemmerer. Attack Languages. In *Proceedings of the IEEE Information Survivability Workshop*, Boston, MA, October 2000.

- [83] G. Vigna, S. T. Eckmann, and R. A. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.
- [84] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.
- [85] G. Vigna, R. A. Kemmerer, and P. Blix. Designing a Highly Configurable Web of Sensors. In Lee et al. [49], pages 69–84.
- [86] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, 2002.