SRI International

REQUIREMENTS AND MODEL FOR IDES—A REAL-TIME INTRUSION-DETECTION EXPERT SYSTEM

Final Report

August 1985

By: Dorothy Denning
Peter G. Neumann

Computer Science Laboratory
Computer Science and Technology Division

Contract No. 83F83-01-00

SRI Project 6169-70

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025-3493
Telephone: (415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-2046
Telex: 334 486

SRI International®

Table of Contents

List of Figures

1. Introduction

The development of a real-time intrusion-detection system is motivated by four factors: (1) most existing systems have security flaws that render them susceptible to intrusions, penetrations, and other forms of abuse; finding and fixing all these deficiencies is not feasible for technical and economic reasons; (2) existing systems with known flaws are not easily replaced by systems that are more secure -- mainly because the systems have attractive features that are missing in the more-secure systems, or else they cannot be replaced for economic reasons; (3) developing systems that are absolutely secure is extremely difficult, if not generally impossible; and (4) even the most secure systems are vulnerable to abuses by insiders who misuse their privileges. Thus, a mechanism that could detect intrusions while they are in progress would be extremely valuable, especially if such a mechanism did not have to know about the particular deficiencies of the target system. Currently, there are no such mechanisms.

The purpose of this report is to lay the groundwork for a real-time intrusion-detection expert system called IDES that could detect security violations independent of whether they are initiated by outsiders who attempt to break into a system or by insiders who attempt to misuse the privileges of their accounts. IDES is based on the hypothesis that exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage. The following examples illustrate (see Appendix I for recent cases of intrusion and Appendix II for examples of known security flaws that can lead to intrusions of the type described here):

- *Attempted break-in* -- If someone attempts to break into a system by trying many passwords with a single login identifier, the number of login failures for the account will increase; if he tries one password with different accounts, the number of login failures for the system as a whole will increase.

- *Masquerading or successful break-in* -- If someone logs into a system through an unauthorized account and password (e.g., obtained by guessing, browsing through someone's desk, or scanning bulletin boards), the login time, location, or connection may differ from that of the account's legitimate user. In addition, the penetrator's behavior on the system may differ considerably from that of the legitimate user; in particular, he might spend

most of his time browsing through file directories and executing system status commands, whereas the legitimate user might concentrate on editing one file or compiling and linking programs. Many break-ins have been discovered by security officers or other users on the system who have noticed the alleged user behaving strangely.

- *Penetration by legitimate user* -- If a user attempts to penetrate the security mechanisms in the operating system, we would expect to see some deviation from his normal use of the system even if his attempt fails; he may execute different programs or trigger more protection violations from attempts to access unauthorized files or programs. If his attempt succeeds, he will have access to commands and files not normally permitted to him.

- *Leakage by legitimate user* -- If a user with access to sensitive documents decides to leak these documents, the user may begin logging into the system at unusual times (e.g., late at night), reading through more files than usual, routing data to remote printers, or generating more hardcopy than usual.

- *Inference by legitimate user* -- If a user with access to a database attempts to obtain unauthorized data from the database using inference techniques, he may retrieve more records than usual.

- *Trojan horse* -- If a Trojan horse is planted in or substituted for a program, its behavior, e.g., CPU time or I/O activity, may change. For example, a Trojan horse login program might write passwords to an obscure file. If a Trojan horse is planted in an experimental library directory that is searched before the standard system directory, there would suddenly be increased activity for the experimental version and decreased activity for the standard version. If a Trojan Horse tries to leak information covertly by allocating available resources or locking out files, the number of resource-exhaustion or file-open exceptions may increase.

- *Virus* -- A virus planted in a system might cause an increase in the frequency of executable files rewritten, storage used by executable files, or a particular program being executed as the virus spreads; or an increase in frequency of file deletions (if the virus is particularly destructive).

- *Denial-of-Service* -- If an intruder is able to monopolize a resource (e.g., network) in such a way that other users are denied access to the resource, activity measured with respect to the intruder or resource may be abnormally high, while activity for the other users may be low.

Unfortunately, the above forms of aberrant usage can also be linked with actions unrelated to security. They could be a sign of a user changing work tasks, acquiring new

skills, or making typing mistakes; software updates; or changing workload on the system. Learning what activity measures discriminate intrusions from other factors will require experimental investigation.

The basic approach of IDES is to monitor system activity as it is recorded in audit records. IDES will examine the audit records as they are generated, update profiles that characterize the behavior of subjects (users) with respect to objects (files, commands, etc), and ascertain whether current activity is abnormal with respect to the profiles. When an anomaly is detected, it will determine whether the security officer should be alerted immediately to a possible intrusion. Periodically, it may also check activity or anomalies accumulated over a time interval. The security officer assists IDES in establishing activities to monitor, but the IDES software and much of its database is system-independent.

IDES monitors the standard operations on a target system: logins, command and program executions, file and device accesses, etc., looking only for deviations in usage. It does not attempt to hunt for specific pattern matches that are known or suspected to exploit a particular security flaw in the system; indeed, it has no knowledge of the target system's security mechanisms or its deficiencies. Although a flaw-based detection mechanism may have some value, it would be considerably more complex and unable to cope with intrusions that exploit deficiencies that are not suspected or with personnel-related vulnerabilities. The approach taken by IDES has the advantage of being system-independent and capable of detecting intrusions when the vulnerability that allowed the intrusion to take place is unknown. By detecting the intrusion, however, IDES may assist the security officer in locating vulnerabilities.

IDES cannot be expected to detect all intrusions. Some intrusions may involve activity that is not monitored by IDES -- e.g., because it is at too low a level in the system. In addition, a person with enough knowledge about IDES may be able to defeat it through gradual modifications of behavior. The goal of IDES is to detect most intrusions and to make it extremely difficult to escape detection.

Although IDES will monitor specific target systems, it will have a built-in system-independent knowledge base that contains templates for generating profiles as well as rules for capturing knowledge about the normal behavior of a system (by updating profiles), detecting abnormal behavior (by statistical tests), and analyzing and reporting abnormalities to the security officer. This system-independent knowledge base, which must be developed by human experts in the field, represents the expert system aspect of IDES.

The remainder of this report is organized as follows: Section 2 states the requirements for IDES. Section 3 presents a model of intrusion detection based on subjects, objects, audit records, profiles, anomaly records, and rules. Section 4 describes and illustrates how the model can be applied to real systems. Section 5 outlines a possible system design for IDES. Section 6 enumerates some research questions. Finally, Section 7 provides our conclusions, namely that the approach appears to be feasible, powerful, and suitable for the subsequent development of a prototype intrusion-detection system.

2. Requirements

We will use the term *system* or *target system* to refer to a computer system (and its applications) that is being monitored for intrusions. The intrusion-detection system itself is generally referred to as IDES. IDES may run on hardware that is physically separate from that of the target system, and may monitor more than one system simultaneously. (IDES might also seek to detect intrusions within its own computer base.)

The requirements of IDES are as follows:

1. *Intrusions detected.* IDES should be able to detect a wide range of possible intrusions, including the following:

    a. *Attempted break-in* -- attempts by outsiders to break into a system, typically by trying different passwords with one or more accounts.

    b. *Masquerading* -- seemingly legitimate accesses resulting from someone having acquired another user's identifier and password, and logging into the system as that user.

    c. *Penetration* -- attempts to circumvent access controls, including password mechanisms, that may be inadequate or incomplete. Two types of penetration are particularly relevant: browsing and tampering.

        - *Browsing*: attempts to violate the desired privacy of user or system information.

        - *Tampering* -- attempts to alter user or system information.

    d. *Second-order access violations* -- attempts to gain unauthorized information through inference or aggregation of collections of individual data items, access to each item of which is authorized. The inference and aggregation security policies may or may not be stated explicitly, and may or may not be enforced by the system. (At present these policies tend to be neither stated nor enforced, but problems resulting therefrom are becoming clearly recognized as important in various contexts.)

    e. *Channels* -- attempts to leak sensitive information from the system or within the system (overtly or covertly), or to exploit existing leakage paths. Two kinds of covert channels are relevant: *storage channels*

(which involve events such as resource exhaustion and exception conditions) and *timing channels* (which involve inferences based on temporal properties of the system). Such violations may or may not involve collaboration between different users.

f. *Denial of service* -- attempts to monopolize or disable resources for the purpose of denying service to other users.

g. *Side-effects* -- injection of viruses, worms, or similar programs that can disrupt service, compromise users, and lead to denial-of-service or serious damage to data and software. Such side-effects may be completely unnoticeable to the victim, but might nevertheless be detectable by IDES (e.g., because of detectable changes in normal system behavior or alterations of programs that have been changed suspiciously).

2. *Applicability.* IDES should be adaptable to different hardware, different operating systems, and different application environments, e.g., C^3I systems, ship-board information systems, general-purpose time-sharing systems, message systems, and networks of computing systems. The system design and software should be application-independent, with all application-dependent information represented in the IDES database rather than in the software.

3. *Discriminating power.* IDES should have the capability of providing a high rate of detection and a low rate of false alarms. This is not straightforward, since adjusting one parameter to increase the detection rate could have the side effect of increasing the false-alarm rate. It will be necessary to determine which activities and statistical measures have the best discriminating power.

4. *Ease of use.* IDES should be considerate of its users, e.g., providing help and prompt facilities, and requiring positive confirmations in cases of irrevocable actions. An example of the latter case would be a request to delete the IDES database, the result of accidentally mistyping one argument. (The system will be used exclusively by trained security officers, which should simplify this requirement somewhat.)

5. *Modifiability.* IDES should have facilities for setting and altering parameters and inserting new rules, but controllably and reliably so that inconsistent changes cannot cause unexpected results (e.g., in setting thresholds contrarily, so that no intrusions would ever be detected). Its output should help the security officer determine which activities and statistical measures have the best discriminating power, so that this information can be incorporated into its parameters and rules.

6. *Self-Learning.* IDES should learn what constitutes normal behavior from its monitoring.

7. *Real-time detection.* To detect intrusions while they are in progress requires that the audit records be made immediately available to IDES for processing, and that IDES itself be able to keep up with the rate at which they are generated. This may limit the level at which auditing can be monitored. Although IDES is designed primarily for real-time detection, the IDES database will also be useful in after-the-fact sleuthing regarding earlier intrusions.

8. *Security of IDES and its database.* The system must enforce privacy[1] and integrity of the IDES database (including nonalterability of primitive information that should remain intact for historical reasons), and restrict denials of service within IDES itself. In addition, the IDES interface used by the monitored system must itself be secure, to prevent spoofing.[2]

---

[1]Audit-trails often contain sensitive items. For example, examination of live data for the MILNET TACACS audit trail [3] shows that the data are littered with passwords, typed out of sync with the TAC login sequence.

[2]Many intrusions that IDES will attempt to detect could represent problems in IDES itself were IDES implemented on the target system. The user community on IDES must be restricted to prevent masquerading, penetration, and tampering. In addition, the possibility of aggregation and inference on the IDES data might require that IDES be classified at a level higher than any of the IDES data. (In practice, this problem has usually either been ignored completely, or else finessed by implicitly giving all security officers higher-than-system-high clearances.) Furthermore, interactive use of IDES should itself be audited, to monitor and/or detect potential misuse by security officers.

3. A Model of Intrusion Detection

A system for intrusion detection can be modeled in terms of six main components:

- *Subjects* -- initiators of activity on a target system.
- *Objects* -- resources managed by the system.
- *Audit records* -- generated by the target system in response to actions performed or attempted by subjects on objects.
- *Profiles* -- structures that characterize the behavior of subjects with respect to objects in terms of statistical metrics and models of observed activity.
- *Anomaly records* -- generated when abnormal behavior is detected.
- *Activity rules* -- actions taken when some condition is satisfied, which update profiles, detect abnormal behavior, and produce reports.

3.1. Subjects and Objects

Subjects are the initiators of actions in the system. A subject is typically a terminal user, but might also be a process acting on behalf of users or groups of users, or might be the system itself. All activity arises through commands initiated by subjects. Subjects may be grouped into different classes (e.g., user groups) for the purpose of controlling access to objects in the system. User groups may overlap.

Objects are the receptors of actions and typically include such entities as files, programs, messages, records, terminals, printers, and user- or program-created structures. When subjects can be recipients of actions (e.g., electronic mail), then those subjects are also considered to be objects in the model. Objects are grouped into classes by type (program, text file, etc.). Additional structure may also be imposed, e.g., records may be grouped into files or database relations; files may be grouped into directories. Different environments may require different object granularity; e.g., for some database applications, granularity at the record level may be desired, whereas for most applications, granularity at the file or directory level may suffice.

3.2. Audit Records

Audit Records are 6-tuples representing actions performed by subjects on objects:

```
<Subject, Action, Object, Exception-Condition, Resource-Usage, Time-stamp>
```

where

- *Action* -- operation performed by the subject on or with the object, e.g., login, logout, read, execute.
- *Exception-Condition*-- denotes which, if any, exception condition is raised on the return. Note that IDES should know the actual exception condition raised by the system, not just the apparent exception condition returned to the subject (which may be censored for security reasons).
- *Resource-Usage* -- list of quantitative elements, where each element gives the amount used of some resource, e.g., number of lines or pages printed, number of records read or written, CPU time or I/O units used, session elapsed time.
- *Time-stamp* -- a time/date stamp that identifies when the action took place. We assume that time-stamps are unique, so that audit records are uniquely identified by them.

We assume that each field is self-identifying, either implicitly or explicitly; e.g., the action field either implies the type of the expected object field or else the object field itself specifies its type. If audit records are collected for multiple systems, then an additional field is needed for a system identifier. For simplicity, we will assume here that IDES monitors only one system.

The target system is responsible for auditing and for transmitting audit records to IDES for analysis. It may also keep an independent audit trail. When IDES receives an audit record, it processes it according to its rules and inserts it in its database.[3] For convenience here, we will imagine that IDES stores all audit records in a single relation Audit-Records (as in a relational database system) with attributes Subject, Action, Object, Exception-Condition, Resource-Usage, and Time-stamp.

---

[3]IDES must, of course, validate each record before incorporating it into its database, to ensure that the database cannot be corrupted by spurious or erroneous records.

Rather than formally modeling a set of operators for manipulating audit records and other structures in the IDES database, we will simply assume that IDES has capabilities of both relational and statistical database systems (though we do not require that an implementation of IDES use a relational database system). A set of audit records is then specified by a selection query of the form:

```
RETRIEVE * FROM Audit-Records WHERE F,
```

where F is any logical formula over the attributes and their domains, and "*" denotes all attributes in the records matching the formula.

Since each audit record specifies a subject and object, it is conceptually associated with some cell in an "audit matrix" whose rows correspond to subjects and columns to objects. Thus, the set of all audit records defined by the query:

```
RETRIEVE * FROM Audit-Records WHERE Subject = s and Object = o ,
```

is associated with row *s*, column *o*. Similar queries define the set of audit records for a row (all records associated with a given subject), column (all records associated with a given object), or arbitrary collection of cells in the matrix. Every statistical measure is computed from audit records associated with one or more cells in the matrix.

The audit matrix is analogous to the "access-matrix" protection model, which specifies the rights of subjects to access objects; that is, the actions that each subject is authorized to perform on each object. The intrusion-detection model of IDES differs from the access-matrix model by substituting the concept of "action performed" (as evidenced by an audit record associated with a cell in the matrix) for "action authorized" (as specified by an access right in the matrix cell). Indeed, IDES observes activity without regard for authorization, implicitly assuming that the access controls in the system permitted an action to occur. Its job is to determine whether activity is unusual enough to suspect an intrusion.

3.3. Profiles

There are two kinds of profiles: activity profiles and profile templates. An *activity profile* characterizes the behavior of a given subject (or set of subjects) with respect to a given object (or set thereof), thereby serving as a *signature* or description of normal activity for its respective subject(s) and object(s). Observed behavior is characterized in terms of a statistical metric and model. A metric is a random variable x representing a quantitative measure accumulated over a period. The period may be a fixed interval of time (minute, hour, day, week, etc.), or the time between two audit-related events (i.e., between login and logout, program initiation and program termination, file open and file close, etc.). From the audit records, IDES obtains observations (sample points) x_i of x, which are used together with a statistical model to determine whether a new observation is abnormal. The statistical model makes no assumptions about the underlying distribution of x; all knowledge about x is obtained from observations. We shall first discuss metrics and models, and then describe the structure of an activity profile. Next, we shall describe profiles for classes of subjects and objects. Finally, we shall discuss how profiles can be generated from *profile templates*.

3.3.1. Metrics

IDES has three types of metrics:

- *Event Counter* -- x is the number of audit records satisfying some property occurring during a period (each audit record corresponds to an event). Examples of event counters are number of logins during an hour, number of times some command is executed during a login session, number of password failures during a minute, and number of file-access violations during a day.

- *Interval Timer* -- x is the length of time between two related events; i.e., the difference between the time-stamps in the respective audit records. Examples are the length of time between successive logins into an account and the length of time between successive executions of some command.

- *Resource Measure* -- x is the quantity of resources consumed by some action during a period as specified in the Resource-Usage field of the audit records. Examples are the total number of pages printed by a user per day, total amount of CPU time consumed by some program during a single execution, and total number of records read per day. Note that a resource measure in IDES is implemented as an event counter or interval timer on the target system. For example, the number of pages printed during a login session is

implemented on the target system as an event counter that counts the number of print events between login and logout, CPU time consumed by a program as an interval timer that runs between program initiation and termination, session elapsed time as an interval timer that runs between login and logout. Thus, whereas event counters and interval timers in IDES measure events at the audit-record level, resource measures acquire data from events on the target system that occur at a level below the audit records. The Resource-Usage field of audit records thereby provides a means of data reduction so that fewer events need be explicitly recorded in audit records.

3.3.2. Statistical Models

Given a metric for a random variable x and n observations x_1, ..., x_n, the purpose of a statistical model of x is to determine whether a new observation x_{n+1} is abnormal with respect to the previous observations. The following models may be included in IDES:

1. *Operational Model.* This model is based on the operational assumption that abnormality can be decided by comparing a new observation of x against fixed limits. Although the previous sample points for x are not used, presumably the limits are determined from prior observations of the same type of variable. The operational model is most applicable to metrics where experience has shown that certain values are frequently linked with intrusions. An example is an event counter for the number of password failures during a brief period, where more than 10, say, suggests an attempted break-in. Another example is an interval timer for the length of time between successive logins into an account, where more than a month, say, suggests a break-in into a "dead" account.

2. *Mean and Standard Deviation Model.* This model is based on the assumption that all we know about x_1, ..., x_n are mean and standard deviation as determined from its first two moments:

$$sum = x_1 + ... + x_n$$

$$sumsquares = x_1^2 + ... + x_n^2$$

$$mean = sum/n$$

$$stdev = sqrt(sumsquares/(n-1) - mean^2) .$$

A new observation x_{n+1} is defined to be abnormal if it falls outside a *confidence interval* that is d standard deviations from the mean for some parameter d:

$$mean \pm d \times stdev$$

By Chebyshev's inequality, the probability of a value falling outside this interval is at most $1/d^2$; for $d = 4$, for example, it is at most .0625.

This model is applicable to event counters, interval timers, and resource measures accumulated over a fixed time interval or between two related events. It has two advantages over an operational model: First, it requires no prior knowledge about normal activity in order to set limits; instead, it learns what constitutes normal activity from its observations, and the confidence intervals automatically reflect this increased knowledge. Second, because the confidence intervals depend on observed data, it allows different limits to be used with different subjects and objects; therefore, what may be normal for one user can be abnormal for another. If all users and objects have the same limits for normal usage, masqueraders, for example, could not be detected.

Note that the moments *sum* and *sumsquares* can be updated dynamically with each occurrence, so that it is unnecessary to keep all values of x to compute mean and standard deviation when a new value arrives. Also, 0 (or null) occurrences should be included so as not to bias the data.

A slight variation on the mean and standard deviation model is to weight the computations, with greater weights placed on more recent values.

3. *Multivariate Model.* This model is similar to the mean and standard deviation model except that it is based on correlations among two or more metrics. This model would be useful if experimental data show that better discriminating power can be obtained from combinations of related measures rather than individually -- e.g., CPU time and I/O units used by a program, login frequency and session elapsed time (which may be inversely related).

4. *Markov Process Model.* This model, which applies only to event counters, regards each distinct type of event (audit record) as a state variable, and uses a state transition matrix to characterize the transition frequencies between states (rather than just the frequencies of the individual states -- i.e., audit records -- taken separately). A new observation is defined to be abnormal if its probability as determined by the previous state and the transition matrix is too low. This model might be useful for looking at transitions between certain commands where command sequences were important.

5. *Time Series Model.* This model, which uses an interval timer together with an event counter or resource measure, takes into account the order and inter-arrival times of the observations x_1, ..., x_n, as well as their values. A new observation is abnormal if its probability of occurring at that time is too low.

A time series has the advantage of measuring trends of behavior over time and detecting gradual but significant shifts in behavior, but the disadvantage of being more costly than mean and standard deviation.

Other statistical models can be considered, for example, models that use more than the first two moments but less than the full set of values.

3.3.3. Profile Structure

An activity profile contains information that identifies the statistical model and metric of a random variable, as well as the set of audit events measured by the variable. The structure of a profile contains 10 components, the first 7 of which are independent of the specific subjects and objects measured:

```
<Variable-Name, Action-Pattern, Exception-Pattern, Resource-Usage-Pattern,
    Period, Variable-Type, Threshold, Subject-Pattern, Object-Pattern, Value>
```

Subject- and Object-Independent Components:

- *Variable-Name* -- name of variable.
- *Action-Pattern* -- pattern that matches zero or more actions in the audit records, e.g., 'login', 'read', 'execute'.
- *Exception-Pattern* -- pattern that matches on the Exception-Condition field of an audit record.
- *Resource-Usage-Pattern* -- pattern that matches on the Resource-Usage field of an audit record
- *Period* -- time interval for measurement, e.g., day, hour, minute (expressed in terms of clock units). This component is null if there is no fixed time interval; i.e., the period is the duration of the activity (e.g., duration of program execution).
- *Variable-Type* -- name of abstract data type that defines a particular type of metric and statistical model, e.g., event counter with mean and standard deviation model.
- *Threshold* -- parameter(s) defining limit(s) used in statistical test to determine abnormality. This field and its interpretation is determined by the statistical model (Variable-Type). For the operational model, it is an upper (and possibly lower) bound on the value of an observation; for the mean and standard deviation model, it is the number of standard deviations from the

mean.

Subject- and Object-Dependent Components:

- *Subject-Pattern* -- pattern that matches on the Subject field of audit records.
- *Object-Pattern* -- pattern that matches on the Object field of audit records.
- *Value* -- value of current (most recent) observation and parameters used by the statistical model to represent distribution of previous values. For the mean and standard deviation model, these parameters are count, sum, and sum-of-squares (first two moments). The operational model requires no parameters.

A profile is uniquely identified by Variable-Name, Subject-Pattern, and Object-Pattern. All components of a profile are invariant except for Value.

Although we leave unspecified the exact format for patterns, we will adapt a notation that borrows constructs from SNOBOL:

Pattern Matching Constructs

'*string*'	string of characters to be matched exactly.
*	wild card matching any sequence of 0 or more characters (wild cards can be embedded in constant strings).
#	match an arbitrary numeric string.
IN(*list*)	match an arbitrary string contained in the set *list*.
$p \rightarrow name$	the string matched by p is immediately associated with *name* for later reference.
$p1\ p2$	match pattern $p1$ followed by $p2$.
$p1 \mid p2$	match pattern $p1$ or $p2$.
$p1\ ,\ p2$	match pattern $p1$ and $p2$.
$\neg p$	match anything but pattern p.

Examples of patterns are:

```
'Smith'

* → User -- match any string and assign result to User

'CPU=' # → Amount --  match string 'CPU=' followed by integer,
                      assign integer to Amount

'<Library>*' -- match files in <Library> directory

¬* -- do not match anything (i.e., match fails)
```

Whenever IDES receives an audit record that matches a variable's patterns, it updates the variable's distribution and checks for abnormality according to the variable's type and anomaly threshold. The distribution of values for a variable is thus derived -- i.e., learned -- as audit records matching the profile patterns are processed.

3.3.4. Profiles for Classes of Subjects and Objects

Profiles can be defined for individual subject-object pairs (i.e., where the Subject and Object patterns match specific names, e.g, Subject 'Smith' and Object 'Foo'), or for aggregates of subjects and objects (i.e., where the Subject and Object patterns match sets of names) as shown in Figure 3-1. For example, file-activity profiles could be created for pairs of individual users and files, for groups of users with respect to specific files, for individual users with respect to classes of files, or for groups of users with respect to file classes. The nodes in the lattice are interpreted as follows:

- *Subject s - Object o*: actions performed by subject s on object o as derived from the matching audit records:

```
RETRIEVE * FROM Audit-Records
     WHERE Subject = s AND Object = o .
```

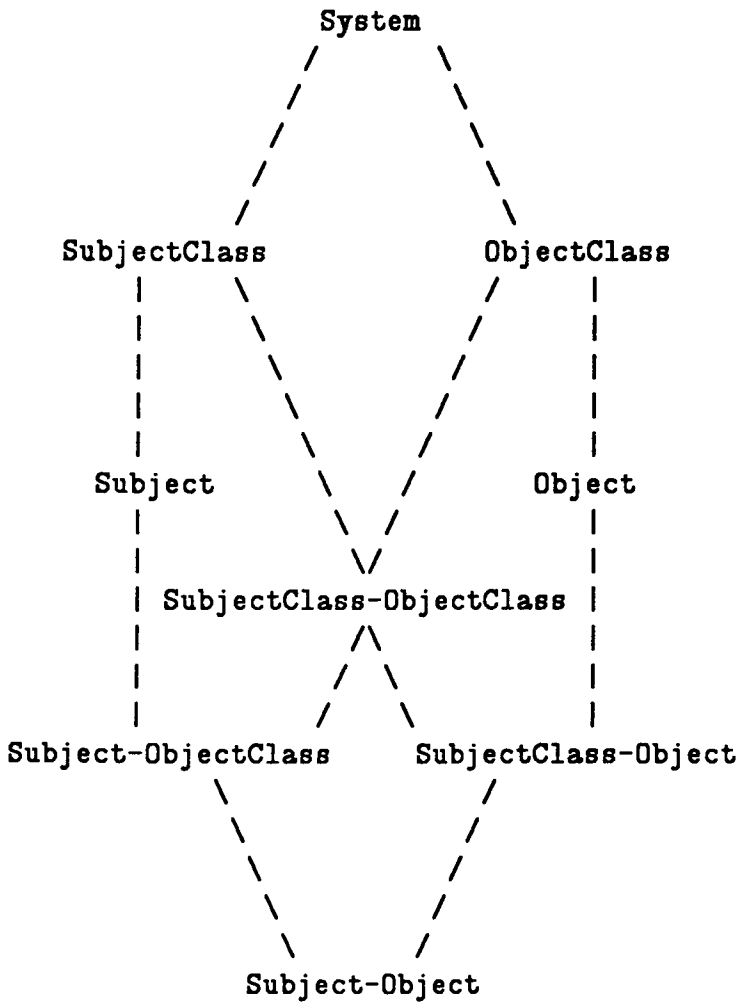
  Examples: user Smith - file Foo, user Jones - file Chess.

- *Subject s - Object Class O*: actions performed by subject s aggregated over all objects in class O as derived from the audit records:

```
RETRIEVE * FROM Audit-Records
     WHERE Subject = s AND Object IN O .
```

  Examples: user Smith - all text files, user Smith - all executable files. The expression "Object IN O" might be represented as a pattern match on a subfield of the Object field that specifies the object's type (class), as a pattern

Figure 3-1: Hierarchy of Subjects and Objects.



match directly on the object's name (e.g., the pattern '<Smith>*' for all files in Smith's library, the pattern '*.EXE' for all executable files), or as a pattern match that tests whether the object is in the set O (e.g., "IN(O)").

- *Subject Class S - Object o*: actions performed on object o aggregated over all subjects in class S as derived from:

```
RETRIEVE * FROM Audit-Records
     WHERE Subject IN S AND Object = o.
```

Examples: privileged users - directory file <Library>, nonprivileged users - directory file <Library>.

- *Subject Class S - Object Class O*: actions aggregated over all subjects in class S and objects in class O as derived from:

```
RETRIEVE * FROM Audit-Records
     WHERE Subject IN S AND Object IN O .
```

Examples: privileged users - system files, nonprivileged users - system files.

- *Subject s*: actions performed by subject s aggregated over all objects as derived from:

```
RETRIEVE * FROM Audit-Records WHERE Subject = s.
```

Examples: user session activity.

- *Object o*: actions performed on object o aggregated over all subjects as derived from:

```
RETRIEVE * FROM Audit-Records WHERE Object = o.
```

Examples: password file activity.

- *Subject Class S*: actions aggregated over all subjects in class S and all objects as derived from:

```
RETRIEVE * FROM Audit-Records WHERE Subject IN S .
```

Examples: privileged user activity, unprivileged user activity.

- *Object Class O*: actions aggregated over all objects in class O and all subjects as derived from:

```
RETRIEVE * FROM Audit-Records WHERE Object IN O.
```

Examples: executable file activity.

- *System*: actions aggregated over all subjects and objects:

```
RETRIEVE * FROM Audit-Records
```

Examples: general system activity

The random variable represented by a profile for a class can aggregate activity for the class in two ways:

- *Class-as-a-whole activity* -- The set of all subjects or objects in the class is treated as a single entity, and each observation of the random variable represents aggregate activity for the entity. An example is a profile for the class of all users representing the average number of logins into the system per day, where all users are treated as a single entity.

- *Aggregate individual activity* -- The subjects or objects in the class are treated as distinct entities, and each observation of the random variable represents activity for some member of the class. An example is a profile for the class of all users characterizing the average number of logins by any one user per day. Thus, the profile represents a 'typical' member of the class.

Whereas class-as-a-whole activity can be defined by an event counter, interval timer, or resource measure for the class, aggregate individual activity requires separate metrics for each member of the class. Thus, it is defined in terms of the lower-level profiles (in the sense of the lattice) for the individual class members. For example, average login frequency per day is defined as the average of the daily total frequencies in the individual user login profiles. A measure for a class-as-a-whole could also be defined in terms of lower-level profiles, but this is not necessary.

The two methods of aggregation serve different purposes with respect to intrusion detection. Class-as-a-whole activity reveals whether some general pattern of behavior is normal with respect to a class. A variable that gives the frequency with which the class of executable program files are updated in the system per day, for example, might be useful for detecting the injection of a virus into the system (which causes executable files to be rewritten as the virus spreads). A frequency distribution of remote logins into the class of dial-up lines might be useful for detecting attempted break-ins.

Aggregate individual activity reveals whether the behavior of a given user (or object) is consistent with that of other users (or objects). This may be useful for detecting intrusions by new users who have deviant behavior from the start.

3.3.5. Profile Templates

When user accounts and objects can be created dynamically, a mechanism is needed to generate activity profiles for new subjects and objects. Three approaches are possible:

1. Manual create -- the security officer explicitly requests IDES to create all profiles for the new subject or object.

2. Automatic explicit create -- IDES automatically generates all profiles for a new user or object when it receives a 'create' record in the audit trail.

3. First use -- IDES automatically generates a profile when a subject (new or old) first uses an object (new or old).

The first approach has the obvious disadvantage of requiring manual intervention on the part of the security officer. The second approach overcomes this disadvantage, but introduces two others. The first is that it does not automatically deal with startup conditions, where there will be many existing subjects and objects. The second is that it requires a subject-object profile to be generated for any pair that is a candidate for monitoring, even if the subject never uses the particular object. This could cause many more profiles than necessary to be generated. For example, suppose file accesses are monitored at the level of individual users and files. Consider a system with 1000 users, where each user has an average of 200 files, giving 200,000 files total and 200,000,000 possible combinations of user-file pairs. If each user accesses at most 300 of those files, however, only 300,000 profiles are needed.

IDES follows the third approach, which overcomes the disadvantages of the others by generating profiles when they are needed from templates. A profile template has the same structure as the profile it generates, except that the subject and object patterns define both a matching pattern (on the audit records) and a replacement pattern (to place in the generated profile). The format for the fields Subject-Pattern and Object-

Pattern is thus:

```
matching-pattern   <-   replacement-pattern
```

where the patterns are defined dynamically during pattern matching. The Value component of a template profile contains the initial values for the variable, as specified by its type.

When a new audit record is received, a process matches the record against both activity profiles and template profiles, obtaining existing profiles and new profiles generated from the matching templates. The subject and object patterns in a generated profile contain the replacement patterns defined during the match; all other fields are copied exactly from the template. If a new profile has the same patterns (for all components) as an existing activity profile, it is discarded; otherwise, it is added to the set of activity profiles. The process then returns the activity profiles matching the audit record.

Separate matching and replacement patterns are needed so that a template can match a wide range of subjects and objects, yet generate a profile for specific subjects, objects, and classes thereof. For examle, consider the following patterns:

```
Subject-Pattern:           * → user          <-  user
Object-Pattern:   IN(Special-Files) → file   <-  file
```

The subject pattern will match any user name and generate a replacement pattern with that name. Similarly, the object pattern will match any file in the list Special-Files and generate a replacement pattern with that name. Now, suppose the list Special-Files contains the file names Password and Accounts. The following shows a sequence of audit records and the profiles that a template with these matching and replacement patterns will generate:

Audit Records		Generated Profiles	
Subject	Object	Subject-Pattern	Object-Pattern
'Smith'	'Password'	'Smith'	'Password'
'Smith'	'Accounts'	'Smith'	'Accounts'
'Jones'	'Accounts'	'Jones'	'Accounts'

The subject and object patterns for a template can be mutually dependent as in following patterns:

```
Subject-Pattern:      * → user   <-    user
Object-Pattern:    '<' user '>*'  <-   '<' user '>*'
```

Here, the object pattern will match any file in the user's directory and generate a profile for the user's directory (if one does not already exist). The following shows a sequence of audit records and the profiles that would be generated from a template containing these patterns:

Audit Records		Generated Profiles	
Subject	Object	Subject-Pattern	Object-Pattern
'Smith'	'<Smith>Game'	'Smith'	'<Smith>*'
'Smith'	'<Smith>Letter'	generated profile is same as above and discarded	
'Jones'	'<Jones>Foo'	'Jones'	'<Jones>*'
'Smith'	'<Jones>Foo	no match, so no profile is generated	

3.4. Anomaly Records

Through its activity rules (next subsection), IDES updates activity profiles and checks for anomalous behavior whenever an audit record is generated or a period terminates. If abnormal behavior is detected, an *anomaly record* is generated having three components:

```
<Event, Time-stamp, Profile>
```

where

- *Event* -- indicates the event giving rise to the abnormality and is either 'audit', meaning the data in an audit record was found abnormal, or 'period', meaning the data accumulated over the current interval was found abnormal.

- *Time-stamp* -- either the time-stamp in the audit record or interval stop time (since we assume that audit records are identified by their time-stamps, this provides a means of tying an anomaly back to an audit record).

- *Profile* -- activity profile with respect to which the abnormality was detected (rather than including the complete profile, IDES might include a 'key' field, which identifies the profile in the database, and the current state of the Value field).

3.5. Activity Rules

An *activity rule* specifies an action to be taken when an audit record or anomaly record is generated, or a time period ends. It consists of two parts: a *condition* that, when satisfied, causes the rule to be 'fired', and a *body*. We will use the term 'body' rather than 'action' to avoid confusion with the actions monitored by IDES. The condition is specified as a pattern match on an event. There are four types of rules:

- *Audit-record rule*, triggered by a match between a new audit record and an activity profile, updates the profile and checks for anomalous behavior.
- *Periodic-activity-update rule*, triggered by the end of an interval matching the period component of an activity profile, updates the profile and checks for anomalous behavior.
- *Anomaly-record rules*, triggered by the generation of an anomaly record, brings the anomaly to the immediate attention of the security officer.
- *Periodic-anomaly-analysis rule*, triggered by the end of an interval, generates summary reports of the anomalies during the current period.

3.5.1. Audit-Record Rules

An audit-record rule is triggered whenever a new audit record matches the patterns in an activity profile. It updates the profile to reflect the activity reported in the record and checks for deviant behavior. If an abnormality is detected, it generates an anomaly record. Since the algorithm for updating the profile and checking for abnormality depends only on the type t of variable (statistical metric and model) represented by the profile, but not on the profile's other components (e.g., subject, object, action, etc.), it can be encoded in a procedure AuditProcess$t$. Thus, all audit record rules are represented by the following generic rule:

```
AUDIT-RECORD RULE
    Condition: new Audit.Record
               Audit.Record matches Profile
               Profile.Variable-Type = t

    Body:      AuditProcesst(Audit-Record, Profile);
END
```

Examples of procedures for AuditProcess$t$ are given in Section 4.3.1.

3.5.2. Periodic-Activity-Update Rules

This type of rule, which is also parameterized by the type t of statistical measure, is triggered whenever a period of length p completes, the Period component of a profile is p, and the Variable-Type component is t. The rule updates the matching profile, checks for abnormal behavior, and generates an anomaly record if an abnormality is detected. (It may also produce a summary activity report.)

```
PERIODIC-VARIABLE-UPDATE RULE
    Condition: Clock mod p = 0
               Profile.Period = p
               Profile.Variable-Type = t

    Body:      PeriodProcesst(Clock, Profile);
END
```

The parameter Clock gives the time at the end of the period.

When the variable represented by the profile for a class is defined in terms of lower-level profiles, as is the case for aggregate individual profiles, the procedure PeriodProcess*t* must obtain data from the individual profiles constituting the class in order to update its Value component. Otherwise, the procedure does not require any data other than what are stored in the profile.

3.5.3. Anomaly-Record Rules

Each anomaly-record rule is triggered whenever a new anomaly record matches patterns given in the rule for its components Event and Profile. Thus, a rule may be conditioned on a particular variable, a particular subject or object, on the audit action that was found to be anomalous, and so forth. For those components of a Profile that are also patterns (e.g., the subject and object components), the patterns given in an anomaly rule must be identical for a match to occur; that is, one pattern matches another only if the patterns are identical. The matching record is brought to the immediate attention of the security officer, with an indication of the suspected type of intrusion. The general form of such a rule is as follows:

```
ANOMALY-RECORD RULE
    Condition: new Anomaly-Record
               Anomaly-Record.Profile matches profile-pattern
               Anomaly-Record.Event   matches event-pattern

    Body:      PrintAlert('Suspect intrusion of type ...', Anomaly-record);
END
```

3.5.4. Periodic-Anomaly-Analysis Rules

This type of rule is triggered by the end of an interval. It analyzes some set of anomaly records for the period and generates a report summarizing the anomalies to the security officer. Its generic form is

```
PERIODIC-ANOMALY-ANALYSIS RULE
    Condition: Clock mod p = 0

    Body:
      Start = Clock - p;
      A = SELECT FROM Anomaly-Records WHERE Anomaly-Record.Time-stamp > Start;
      generate summary report of A;
END
```

The rule selects all anomaly records belonging to the period from the set (relation) of all anomaly records, Anomaly-Records.

To facilitate the reporting of anomalies, the model might be enhanced to include anomaly profiles. An anomaly profile would be similar to an activity profile except that updates would be triggered by the generation of an anomaly record within IDES rather than an audit record from the target system. Whether such a structure would be useful, however, is unclear.

4. Applying the Model

This section describes and illustrates application of the model to real systems. Subsection 4.1 discusses several problems associated with generating audit records for real-time intrusion detection and illustrates how the activity on a system can be represented by a series of audit records. Subsection 4.2 gives definitions for different types of activity variables and complete descriptions of profiles that use these types to measure activity related to login and session activity, command and program execution, and file and database access. Subsection 4.3 shows how the generic rules are instantiated to specific types of statistical measures.

4.1. Auditing

This section discusses how complex system activities can be represented by audit records, when audit records should be generated, and deficiencies of existing audit mechanisms. An example illustrating how activity on a system is audited is given at the end.

4.1.1. Complex Operations on Multiple Objects

Most operations on a system involve multiple objects. For example, file copying involves the copy program, the original file, and the copy. Editing involves the edit program plus one or more files. Compiling involves the compiler, a source program file, an object program file, and possibly intermediate files and additional source files referenced through "include" statements. Linking involves the linker, object files, library files, an executable file, and a map file. Sending an electronic mail message involves the mail program, possibly multiple destinations in the "To" and "cc" fields, and possibly "include" files.

After some consideration, we decided to decompose all activity into single-object actions so that each audit record references only one object. Thus, for example, file copying is decomposed into an execute operation on the copy command, a read operation on the source file, and a write operation on the destination file. There are three reasons for this decomposition. First, since objects are the protectable entities of a system, the decomposition is consistent with the protection mechanisms of systems. As noted earlier,

whereas access controls aim to restrict what actions a subject is allowed to perform on a given object in accordance with the protection policies of the system, IDES monitors what actions are attempted and which succeed. IDES can potentially discover both attempted subversions of the access controls (by noting an abnormality in the number of exception conditions returned) and successful subversions (by noting an abnormality in the set of objects accessible to the subject). Second, single-object audit records greatly simplify the model and its application. Third, the audit records produced by existing systems generally contain a single object, though some systems provide a way of linking together the audit records associated with a "job step" (e.g., copy or compile) so that all files accessed during execution of a program can be identified. Although it appears that the combinatorial information lost by decomposition will not interfere with our ability to detect intrusions, we leave open the possibility of needing a more complex model to handle multi-object actions.

4.1.2. Time of Audit

The time at which audit records are generated determines what type of data is available. If the audit record for some action is generated at the time an action is requested, it is possible to measure successful and unsuccessful attempts (e.g., because of protection violations) to perform the activity, even if the action should abort or cause a system crash. If it is generated when the action completes, it is possible to measure the resources consumed by the action and exception conditions that may cause the action to terminate abnormally (e.g., because of resource overflow). Thus, auditing an activity after it completes has the advantage of providing more information, but the disadvantage of not allowing immediate detection of abnormalities, especially those related to break-ins and system crashes. Thus, activities such as login, execution of high risk commands (e.g., to acquire special "superuser" privileges), or access to sensitive data should be audited when they are attempted so that penetrations can be detected immediately; if resource-usage data are also desired, additional auditing can be performed on completion as well. For example, access to a database containing highly sensitive data may be monitored when the access is attempted and then again when it completes to report the number of records retrieved or updated. Most existing audit systems monitor session activity at both initiation (login), when the time and location of

login are recorded, and termination (logout), when the resources consumed during the session are recorded. They do not, however, monitor both the start and finish of command and program execution or file accesses. IBM's System Management Facilities (SMF) [6], for example, audit only the completion of these activities.

4.1.3. Deficiencies of Existing Audit Mechanisms

The auditing mechanisms of existing systems generally provide the types of records desired, but may not monitor all of the behavior of potential interest. For example, Berkeley 4.2 UNIX[4] [8] monitors command usage but not file accesses or file protection violations. Some systems do not record all login failures. Most systems do not monitor the activity of the login program itself (in terms of I/O and CPU activity), possibly because the resources it consumes are not charged to any user's account. Programs, including system programs, that are not invoked at the command level are not explicitly monitored (their activity is included in that for the main program). The extent to which a system should be modified to audit additional activity is unclear and requires further investigation.

Deficiencies in the record structures are also present. Most SMF audit records, for example, do not contain a subject field; the subject must be reconstructed by linking together the records associated with a given job. Time-stamps are not always unique; indeed, they provide the means of linking SMF audit records for a batch job. Requiring unique time-stamps, however, is not essential if audit records can be uniquely identified by other fields. Protection violations are sometimes provided through separate record formats rather than as an exception condition in a common record; VM password failures at login, for example, are handled this way (there are separate records for successful logins and password failures).

Another problem with existing audit records is that they contain little or no descriptive information to identify the values contained therein. Every record type has its own structure, and the exact format of each record type must be known to interpret

---

[4]UNIX is a trademark of Bell Laboratories.

the values. A uniform record format with self-identifying data would be preferable so that the IDES software to process the audit records could be system-independent. This could be achieved either by modifying the software that produces the audit records in the target system, or by writing a filter that translates the records into a standard IDES format.

4.1.4. Sample Audit

To see how the actions initiated by users can be represented as audit records, consider the scenario in Figure 4-1, which shows Smith inserting a preconceived Trojan horse into a game program, recompiling the program, trying unsuccessfully to install it in the main <Library> directory, settling for an experimental library <LibraryExp>, and sending mail to two other users. A user whose search path specifies the experimental library before the main library would automatically get the intentionally flawed GAMES program rather than the original one. Independent activity of two other users, Brown and a would-be interloper, are also shown.

A sequence of audit records for the scenario is shown in Figure 4-2. For simplicity, we have omitted the Resource-Usage field; thus, the structure of an audit record is

```
(Subject, Action, Object, Exception-Condition, Time-stamp)
```

We have generated all audit records at the time the activity is attempted, except for session activity, which is recorded both at login and logout. (In the Exception-Condition field, the string "0" denotes the absence of a raised exception condition.)

Figure 4-1: Hypothetical Sequence of Activity.

COMMAND SEQUENCE	SYSTEM RESPONSE
{BY SMITH}	
login Smith password	Previous login, failed attempts
edit	
read <Library>GAME.ADA	Read GAME into edit buffer
read TROJAN.HORSE	Insert Trojan horse
write GAME.ADA	''<Smith>GAME.ADA written''
exit edit	
Ada GAME.ADA {compile}	''GAME.EXE created''
copy GAME.EXE <Library>GAME.EXE	''No write access''
copy GAME.EXE <LibraryExp>GAME.EXE	
mail	
send to Jones cc Green@USC-ISI	
exit mail	
logout	Time elapsed, charges
{BY BROWN WHILE SMITH LOGGED IN}	
login Brown password1	Previous login, failed attempts
connect Smith password2	''Incorrect Password''
directory <Smith>	Lists Smith's files
read <Smith>TROJAN.HORSE	''No read access''
link Smith	''Protection Violation''[5]
logout Smith	''Protection Violation''[6]
logout	Time elapsed, charges
{BY UNIDENTIFIED USER ATTEMPTING TO LOGIN AS JONES}	
finger {or *who?*}	''Please log in.''[7]
login JONES password3	''Login not permitted''
login JONES password4	''Login not permitted''
login JONES password5	System hangs up on the user.

[5]Smith had refused links. This would have been "User Not Logged In" if Smith had already logged out.

[6]Permitted only for superuser or another logged-in job of Smith.

[7]Some systems permit certain commands prior to *login* – in which case the response here would be a list of connected users. However, this practice is risky if it provides would-be penetrators with any helpful information.

Figure 4-2: Audit Records for Activity of Figure 4-1.

```
(Smith, login, hard-wired-terminal A, 0, t0)     {Smith Login}
(Smith, execute, <Library>EDIT.EXE,   0, t1)     {Edit}
(Smith, read,    <Library>GAME.ADA,   0, t2)
(Smith, read,    <Smith>TROJAN.HORSE, 0, t3)
(Smith, write,   <Smith>GAME.ADA,     0, t4)     {Save Trojan horse game}
(Smith, execute, <Library>ADA.EXE,    0, t5)     {Compile}
(Smith, read,    <Smith>GAME.ADA,     0, t6)
(Smith, write,   <Smith>GAME.EXE,     0, t7)
(Smith, execute, <Library>COPY.EXE,   0, t8)     {Try to Copy to Library}
(Smith, read,    <Smith>GAME.EXE,     0, t9)
(Smith, write,   <Library>GAME.EXE, write-viol, t10)
(Smith, execute, <Library>COPY.EXE,   0, t11)    {Copy to LibraryExp}
(Smith, read,    <Smith>GAME.EXE,     0, t12)
(Smith, write,   <LibraryExp>GAME.EXE,0, t13)
(Smith, execute, <Library>MAIL.EXE,   0, t14)    {Send mail}
(Smith, send,    Jones,               0, t15)
(Smith, send,    Green@USC-ISI,       0, t16)
(Brown, login,   dial-up-port X,      0, t17)    {Brown login}
(Brown, connect, Smith,          bad-PW, t18)    {Try to connect to Smith}
(Brown, execute, <Library>DIR.EXE,    0, t19)    {Read directory}
(Brown, read,    <Smith>,             0, t20)
(Brown, read,    <Smith>TROJAN.HORSE, read-viol, t21)[8]
(Brown, link,    Smith,    link-refused, t22)
(Brown, execute, logout another user, privilege-viol, t23)
(Brown, logout,  dial-up-port X,      0, t24)    {Brown logout}
(Smith, logout, hard-wired-terminal A,0, t25)    {Smith logout}
(Unknown, execute, <Library>FINGER.EXE, not-logged-in,     t26)[9]
(Jones, login,   ARPANET socket X USC-ISI, bad-PW,         t27)
(Jones, login,   ARPANET socket X USC-ISI, bad-PW,         t28)
(Jones, login,   ARPANET socket X USC-ISI, bad-PW-hang-up, t29)
```

[8]Note a potential security glitch in the operating system that permits Brown to see an entry for TROJAN.HORSE in Smith's directory, but not to be able to access the file.

[9]Since the Subject is unknown in such cases, the User ID should be that of a collective Unknown user -- although it might be useful to use a finer-grain subject, such as Unknown-dial-up or Unknown-hard-wired. Alternatively, if all commands other than login are forbidden prior to login, the problem goes away.

4.2. Activity Profiles

This section illustrates how profiles as described is Section 3.3 are applied to measure login and session activity, command and program usage, file accesses, and database activity. We shall first give complete definitions for variable types that represent the three different metrics (event counters, interval timers, and resource measures) and simple statistical models, and then give profile specifications in terms of these types.

4.2.1. Variable Types

Figures 4-4 through 4-9 define six abstract data types in terms of their structure and operations. The first two types are based on the operational model, where a fixed bound determines abnormality. The remaining four types are based on the mean and standard deviation model, where a confidence interval determines abnormality.

- SimpleCounter -- event counter over a fixed-length interval based on the operational model.
- IntervalTimer -- interval timer based on the operational model.
- EventCounter -- event counter over a fixed-length interval based on the mean and standard deviation model.
- EventCounterByDayAndHour -- array of event counters by day of week and hour of day based on the mean and standard deviation model.
- ResourceByActivity -- resource measure accumulated over duration of action based on the mean and standard deviation model.
- ResourceByTime -- resource measure accumulated over fixed-length interval based on the mean and standard deviation model.

For each type *t*, we define three procedures: "New*t*" for processing a new observation of the variable, "Update*t*Distrib" for updating the distribution of the variable to include the new observation (or simply "Update*t*" if there is no distribution), and "Check*t*" for testing whether the new observation is abnormal. The syntax for the structures and procedures is reminiscent of both Pascal and C. Since moments are used to represent the distributions of the first three types, a subtype Moments is defined in

to represent the distributions of the first three types, a subtype Moments is defined in Figure 4-3.

All six types represent a single metric defined in terms of a single profile. They could be used for measuring individual subject-object activity or class-as-a-whole activity. Multivariate measures, or measures for aggregate individual activity (which are used to define 'typical' members of a class) could be defined in terms of these more primitive metrics. For example, multivariate profiles representing the correlation between login frequency and total session elapsed time during a day could be defined in terms of single-variable profiles for login frequency and session elapsed time. The multivariate profiles would be updated at the end of the day using the values in the more primitive profiles. Similarly, profiles for aggregate individual activity would be periodically updated from lower-level profiles for the class members.

Figure 4-3: Subtype Moments.

```
TYPE Moments = RECORD
  {Represent distribution characterized by mean & standard deviation
   derived from the first two moments.
  }
   count:      integer = 0;  {number of occurrences of variable}
   sum:        integer = 0;  {sum of values}
   sumsquares: integer = 0;  {sum of squares of values}
END


UpdateMoments(VAR x: Moments, new:integer)  {update distribution with new value}
  BEGIN
    x.count += 1;
    x.sum += new;
    x.sumsquares += x.current**2;
  END


AbnormalMoments(x: Moments, new: integer, threshold: integer)
   {check if new value exceeds the mean by more than threshold deviations --
    one might also check whether it is below the mean by that amount}
  BEGIN
    IF (x.count <= 1) THEN RETURN(false);
    mean = x.sum/x.count;
    stdev = sqrt(x.sumsquares/(x.count - 1) - mean**2);
    diff = (new - mean)/stdev;
    IF (diff > threshold) THEN RETURN(true) ELSE RETURN(false);
  END
```

Figure 4-4: Simple Event Counter, Operational Model.

```
TYPE SimpleCounter = RECORD
   counter:    integer = 0;  {current count}
END


NewSimpleCounter(x: SimpleCounter)
  BEGIN
    x.counter += 1;
  END


UpdateSimpleCounter(x: SimpleCounter)   {reset counter}
  BEGIN
    x.counter = 0;
  END


CheckSimpleCounter(x: SimpleCounter, threshold)  {check if current value is too large}
  BEGIN
    IF (x.counter > threshold) THEN RETURN(true) ELSE RETURN(false);
  END
```

Figure 4-5: Interval Counter, Operational Model.

```
TYPE IntervalTimer = RECORD

   interval:    integer = 0;  {length of time since last occurrence}
   last:        integer = 0;  {time of last occurrence}
END


NewIntervalTimer(var x: IntervalTimer, time: integer)
   {update last time of occurrence}
  BEGIN
    x.interval = time - x.last;
    x.last = time;
  END


UpdateIntervalTimer(var x: IntervalTimer, time: integer)  {reset}
  BEGIN
    x.interval = 0;
    x.last = time;
  END


CheckIntervalTimer(x: IntervalTimer, threshold: integer)
   {check if delay since last occurrence is too long}
  BEGIN
    IF (x.interval > threshold) THEN RETURN(true); ELSE RETURN(false);
  END
```

Figure 4-6: Event Counter, Mean and Standard Deviation Model.

```
TYPE EventCounter = RECORD

   distribution: Moments;
   counter:      integer = 0;  {current value, initially 0}
END


NewEventCounter(VAR x: EventCounter)  {new event occurs}
  BEGIN
    x.counter += 1;
  END


UpdateEventCounterDistrib(VAR x: EventCounter)
   {add current counter to distribution and reset counter}
  BEGIN
    UpdateMoments(x.distribution, x.counter);
    x.counter = 0;        {reset to 0}
  END


CheckEventCounter(x: EventCounter, threshold: integer)   {abnormality test}
   {check if current counter is more than threshold deviations from mean}
  BEGIN
    IF AbnormalMoments(x.distribution, x.counter, threshold)
      THEN RETURN(true) ELSE RETURN(false);
  END
```

Figure 4-7: Event Counter By Day and Hour, Mean and Standard Deviation Model.

```
TYPE EventCounterByDayAndHour = RECORD

   distribution: array[1:7,1:24] of Moments;
   counter:      integer = 0;  {value for current period, initially 0}
END


NewEventCounterByDayAndHour(VAR x: EventCounter)  {new event occurs}
  BEGIN
    x.counter += 1;
  END


UpdateEventCounterByDayAndHourDistrib(VAR x: EventCounter,
                                      day: integer, hour: integer)
   {add current counter to distribution and reset counter}
  BEGIN
    UpdateMoments(x.distribution[day,hour], x.counter);
    x.counter = 0;        {reset to 0}
  END


CheckEventCounterByDayAndHour(x: EventCounter, day: integer, hour: integer,
                              threshold: integer)
   {check if current counter is more than threshold deviations from mean}
  BEGIN
    IF AbnormalMoments(x.distribution[day,hour], x.counter, threshold)
      THEN RETURN(true) ELSE RETURN(false);
  END
```

Figure 4-8: Resource Measure over Activity, Mean and Standard Deviation Model.

```
TYPE ResourceByActivity = RECORD

   distribution: Moments;
   amount:       integer = 0;  {resources used during most recent action}
END


NewResourceByActivity(VAR x: ResourceByActivity, new: integer)  {new amount}
  BEGIN
    x.amount = new;
  END


UpdateResourceByActivityDistrib(VAR x: ResourceByActivity)
   {add current amount to distribution}
  BEGIN
    UpdateMoments(x.distribution, x.amount);
    x.amount = 0;
  END


CheckResourceByActivity(x: ResourceByActivity, new: integer, threshold: integer)
   {check if current amount is more than threshold deviations from mean}
  BEGIN
    IF AbnormalMoments(x.distribution, x.amount, threshold)
      THEN RETURN(true) ELSE RETURN(false);
  END
```

Figure 4-9: Resource Measure over Time, Mean and Standard Deviation Model.

```
TYPE ResourceByTime = RECORD

   distribution: Moments;
   amount:       integer = 0; {resources used in current time interval}
END


NewResourceByTime(VAR x: ResourceByTime, new: integer)  {new amount}
  BEGIN
    x.amount += new;        {add to amount current time period}
  END


UpdateResourceByTimeDistrib(VAR x: ResourceByTime)
   {add current amount to distribution}
  BEGIN
    UpdateMoments(x.distribution, x.amount);
    x.amount = 0;
  END


CheckResourceByTime(x: ResourceByTime, new: integer, threshold: integer)
   {check if current amount is more than threshold deviations from mean}
  BEGIN
    IF AbnormalMoments(x.distribution, x.amount, threshold)
      THEN RETURN(true) ELSE RETURN(false);
  END
```

4.2.2. Login and Session Profiles

We now consider profile templates for generating activity profiles related to login and session activity. Login and session activity is represented in audit records where the subject is a user, the object is the user's login location (terminal, workstation, network, remote host, port, etc., or some combination), and action is 'login' or 'logout'. Locations may be grouped into classes by properties such as type of connection: hard-wired, dial-up, network, etc. or type of location: dumb terminal, intelligent workstation, network host, etc.

Before describing specific profile templates, we consider several possibilities for their subject and object patterns:

- Individual user and location:

```
Subject-Pattern:    * → user  <-  user
Object-Pattern:     * → loc   <-  loc
```

  Subject-Pattern specifies that whatever name is matched in the Subject field of an audit record, e.g., 'Smith', should be copied into the Subject-Pattern for the generated activity profile. Object-Pattern similarly causes exact location names to be placed in generated activity profiles.

- Individual user, all locations grouped together:

```
Subject-Pattern:    * → user  <-  user
Object-Pattern:           *     <-   *
```

  Because both the matching and replacement patterns for objects specify wild cards, the template will match any object and generate a profile that matches any object.

- User groups, all locations:

```
Subject-Pattern:  IN(user-group) <- IN(user-group)
Object-Pattern:                *   <-  *
```

  where *user-group* is a list containing the names of all users in the group. A template is defined for each such group to generate the activity profiles for the group.

- All users, all locations:

```
Subject-Pattern:        *   <-   *
Object-Pattern:         *   <-   *
```

A template with these patterns generates a single profile representing all users and locations.

- Individual user, locations grouped by connection type:

```
Subject-Pattern:          * → user     <-  user
Object-Pattern:      connection-type  <-  connection-type
                             *        <-  *
```

where *connection-type* is DIAL-UP, HARD-WIRED, etc. We assume in this example that the object component in an audit record has the substructure (*connection-type*, *location-name*), so separate patterns are given for each of its elements. A template is defined for each connection type to generate the activity profiles for its class.

The following gives the subject- and object-independent components of template profiles for various measures of login and session activity (see Section 3.3.3 for a description of the components):

- *LoginFrequency* -- event counter that measures login frequency by day and time using the mean and standard deviation model. Since a user's login behavior may vary considerably during a work week, login occurrences may be represented by an array of event counters parameterized by day of week (specific day or weekday v. weekend) and time of day (hour or shift) (Another possible breakdown is: weekday, evening, weekend, night.)

```
Variable-Name:           LoginFrequency
Action-Pattern:          'login'
Exception-Pattern:       0                {successful login}
Resource-Usage-Pattern:
Period:                  hour             {number of clock units per hour}
Variable-Type:           EventCounterByDayAndHour
Threshold:               4                {standard deviations from mean}
```

Profiles for login frequencies may be especially useful for detecting masqueraders, who are likely to log into an unauthorized account during off-hours when the legitimate user is not expected to be using the account. Although they could be defined for individual users and specific locations, login-frequency profiles may be more suitable for classes of locations -- either all locations taken together or aggregated by type of location or connection. A system with 1000 users and 20 groups might have 1000 profiles that are

user-specific, 20 profiles that aggregate over each group, and 1 profile that aggregates over everyone, giving a total of 121 profiles if all locations are aggregated together. If locations are aggregated into two groups, say, there would be 242 profiles.

- *LocationFrequency* -- event counter that measures the frequency of login at different locations using the mean and standard deviation model. Although this measure could be broken down by day of week and time of day (since a user may login from one location during normal working hours and another during non-working hours), we will represent it here as a single counter where the period is one day:

```
Variable-Name:          LocationFrequency
Action-Pattern:         'login'
Exception-Pattern:      0           {successful login}
Resource-Usage-Pattern:
Period:                 day
Variable-Type:          EventCounter
Threshold:              4           {standard deviations from mean}
```

Because the variable relates to specific objects, it should be defined for individual locations or location types. It may be useful for detecting masqueraders -- e.g., if someone logs into an account from a location that the legitimate user never uses, or penetration attempts by legitimate users -- e.g., if someone who normally works from an unprivileged local terminal logs in from a highly privileged terminal.

- *LastLogin* -- interval timer measuring time since last login using the operational model.

```
Variable-Name:          LastLogin
Action-Pattern:         'login'
Exception-Pattern:      0           {successful login}
Resource-Usage-Pattern:
Period:
Variable-Type:          IntervalTimer
Threshold:              50 days     {fixed limit}
```

This type of profile could be defined for individual users but location classes, since the exact location seems less relevant than the lapse of time. It would be particularly useful for detecting a break-in on a "dead" account.

- *SessionElapsedTime* -- resource measure of elapsed time per session based on mean and standard deviation model:

```
Variable-Name:          SessionElapsedTime
Action-Pattern:         'logout'
Exception-Pattern:      0           {successful logout}
Resource-Usage-Pattern: 'ElapsedTime=' # → amount
Period:                             {collect on session basis}
Variable-Type:          ResourceByActivity
Threshold:              4           {deviations from mean}
```

The Resource-Usage-Pattern means matches a string that contains the sequence 'ElapsedTime=' followed by a number (which matches the '#' and is associated with 'amount'). This type of profile could be defined for individual users or groups, but object classes. Deviations might signify masqueraders.

- *SessionOutput* -- resource measure of quantity of output to terminal per session using mean and standard deviation model (output might also be measured on a daily basis):

```
Variable-Name:          SessionOutput
Action-Pattern:         'logout'
Exception-Pattern:      0                   {successful logout}
Resource-Usage-Pattern: 'SessionOutput=' # → amount
Period:                                     {collect on session basis}
Variable-Type:          ResourceByActivity
Threshold:              4                   {standard deviations}
```

Defining this type of profile for individual locations or classes thereof may be useful for detecting excessive amounts of data being transmitted to remote locations, which could signify leakage of sensitive data.

- *SessionCPU, SessionIO, SessionPages, etc.* -- resource measures accumulated on a daily bases (or session basis) based on the mean and standard deviation model. The following shows the structure of SessionCPU; the others are similar.

```
Variable-Name:          SessionCPU
Action-Pattern:         'logout'
Exception-Pattern:      0               {successful logout}
Resource-Usage-Pattern: 'CPU=' # → amount
Period:                 day
Variable-Type:          ResourceByTime
Threshold:              4               {deviations from mean}
```

These profiles may be useful for detecting masqueraders.

- *PasswordFails* -- event counter that measures password failures at login using the operational model:

```
Variable-Name:          PasswordFails
Action-Pattern:         'login'
Exception-Pattern:      'bad-PW'
Resource-Usage-Pattern:
Period:                 5 minutes
Variable-Type:          SimpleCounter
Threshold:              10              {fixed bound}
```

This type of profile is extremely useful for detecting attempted break-ins, and should be defined both for individual users and all users taken together. An attack involving many trial passwords on a particular account would show up as an abnormally high number of password failures with respect to a profile for the individual account (user); an attack involving a single trial password over many accounts would show up as an abnormally high number of password failures with respect to a profile for all users. Recording password failures by location class may be desirable if there are substantially more failures across dial-up lines caused by unreliable communications. Password failures might be recorded over a fairly short period of time, say at most a few minutes, since break-ins are usually attempted in a burst of activity.

- *LocationFails* -- event counter measuring failures to login from specified terminals based on operational model:

```
Variable-Name:          LocationFails
Action-Pattern:         'login'
Exception-Pattern:      'illegal-location'
Resource-Usage-Pattern:
Period:
Variable-Type:          SimpleCounter
Threshold:              1               {fixed bound}
```

This type of profile might be defined for individual users, but aggregates of locations together since the exact location is less significant than that it was unauthorized. It may be used to detect attempted break-ins or attempts to log into highly privileged terminals.

4.2.3. Command or Program Execution Profiles

Command or program execution activity is represented in audit records where the subject is a user, the object is the name of a program (for simplicity, we will assume that all commands are programs and not distinguish between the two), and action is 'execute'. Programs may be classified and aggregated by whether they are privileged (executable only by privileged users or in privileged mode) or nonprivileged, by whether they are system programs or user programs, by whether they are typically used by novices or experts, or by some other property.

Since subject and object patterns for program template profiles are similar to those defined for session profiles, we will not give a complete list of possibilities. The following, however, use patterns that are unlike those described for login and session activity:

- Individual users, all programs in system library directory grouped together:

```
Subject-Pattern:    * → user    <-  user
Object-Pattern:    '<Library>*'  <-  '<Library>*'
```

- Individual users, all programs in user's directory grouped together:

```
Subject-Pattern:     * → user    <-   user
Object-Pattern:    '<' user '>*'  <-   '<' user '>*'
```

- Individual users, all programs in all other directories grouped together:

```
Subject-Pattern:              * → user                <-  user
Object-Pattern:    ('<' ¬(user | 'Library') '>*')  <-
                   ('<' ¬(user | 'Library') '>*')
```

We now describe statistical measures for program profiles:

- *ExecutionFrequency* -- event counter measuring the number of times a program is executed during some time period using the mean and standard deviation model:

```
Variable-Name:           ExecutionFrequency
Action-Pattern:          'execute'
Exception-Pattern:       0
Resource-Usage-Pattern:
Period:                  day
Variable-Type:           EventCounter
Threshold:               4              {standard deviations}
```

This type of profile may be defined for individual users and programs or classes thereof. A profile for individual users and commands may be useful for detecting masqueraders, who are likely to use different commands from the legitimate users; or for detecting a successful penetration by a legitimate user, who will then have access to privileged commands that were previously disallowed. A profile for individual programs but all users may be useful for detecting substitution of a Trojan horse in an experimental library that is searched before the standard library, since the frequency of executing the original program would drop off. It may also be useful for detecting viruses where the virus is manifested in a program invoked by infected programs.

- *ProgamCPU, ProgramIO, etc.* -- resource measures per execution of a program using the mean and standard deviation model. A profile for ProgramCPU is given by:

```
Variable-Name:           ProgramCPU
Action-Pattern:          'execute'
Exception-Pattern:       0
Resource-Usage-Pattern:  'CPU=' # → amount
Period:
Variable-Type:           ResourceByActivity
Threshold:               4                    {standard deviations}
```

  This type of profile may be defined for individual users and programs or classes thereof. An abnormal value for one of these measures applied to the aggregate of all users might suggest injection of a Trojan horse or virus in the original program, which performs side-effects that increase its I/O or CPU usage. Rather than accumulating data on a per execution basis, the variables could measure activity accumulated over some time interval, e.g., a day. A fixed-length interval, however, seems less useful here for intrusion detection.

- *ExecutionDenied* -- event counter for number of attempts to execute an unauthorized program during a day based on operational model:

```
Variable-Name:           ExecutionDenied
Action-Pattern:          'execute'
Exception-Pattern:       'privilege-viol'
Resource-Usage-Pattern:
Period:                  day
Variable-Type:           SimpleCounter
Threshold:               3              {fixed bound}
```

  Defining this type of profile for individual users might be useful for detecting a penetration attempt by some particular user. This type of profile might also be defined for individual programs that are highly sensitive, in which case a threshold of 1 may be appropriate.

- *ProgramResourceExhaustion* -- event counter for the number of times a program terminates abnormally during a day because of inadequate resources using the operational model:

```
Variable-Name:           ProgramResourceExhaustion
Action-Pattern:          'execute'
Exception-Pattern:       'resource-exhaust'
Resource-Usage-Pattern:
Period:                  day
Variable-Type:           SimpleCounter
Threshold:               5              {fixed bound}
```

  This type of profile might be defined for individual programs or classes of

programs to detect a program that consistently aborts (e.g., because it is leaking data to the user through a covert channel based on resource usage).

It is probably unnecessary, even if feasible, to keep individual profiles for all programs in the system. Individual profiles for some programs seem essential, however, to detect masquerading, attempted penetrations, and Trojan horse substitutions from command usage (these profiles could, however, aggregate over all users). Programs that might be monitored individually include those that are privileged, security relevant, indicative of browsing, or frequently used. Examples are:

link or write to another user (2 in the scenario)
connect or change working directory (t18 in the scenario)
logout another user (3 in the scenario)
create a new user account
change password
change access privileges for a user, especially authorizations for "superuser" status
delete an account
status commands that tell what other users are doing (e.g., *ps* on UNIX, *sys* on TOPS-20, and *who* and *finger*) (6 in the scenario)
editors, document formatters, compilers, linkers, software tools, utilities, mail programs, games

4.2.4. File-Access Profiles

File-access activity is represented in audit records where the subject is a user, the object is the name of a file, and action is 'read', 'write', 'create', or 'delete' ('append' is also a possibility for some systems). Files may be classified by type: text, executable program, directory, etc.; by whether they are system files or user files; or by some other property. Since a program is a file, it can be monitored both with respect to its execution activity and its file-access activity.

Subject and object patterns for file template profiles may be similar to those defined for session and program profiles. The following example further illustrates how files might be aggregated into classes:

- Individual users, files grouped by the extension (type):

```
Subject-Pattern:   * → user  <-  user
Object-Pattern:     '*.ext'  <-  '*.ext'
```

where *ext* is 'EXE', 'TXT', 'ADA', etc.

The following measures are candidates for profiles:

- *ReadFrequency, WriteFrequency, CreateFrequency, DeleteFrequency* -- event counters that measure the number of accesses of their respective types during a day (or some other period) using the mean and standard deviation model. The following shows the components of a profile for ReadFrequency:

```
Variable-Name:           ReadFrequency
Action-Pattern:          'read'
Flag-Pattern:
Resource-Usage-Pattern:
Period:                  day
Variable-Type:           EventCounter
Threshold:               4          {standard deviations from mean}
```

Read and write access frequency profiles may be defined for individual users and files or classes thereof. Create and delete access profiles, however, only make sense for aggregate file activity since any individual file is created and deleted at most once. Abnormalities for read and write access frequencies for individual users may signify masquerading or browsing. They may also indicate a successful penetration, since the user would then have access to files that were previously disallowed.

- *RecordsRead, RecordsWritten* -- resource measures for the number of records read or written per access (measurements could also be made on a daily basis) using the mean and standard deviation model:

```
Variable-Name:           RecordsRead
Action-Pattern:          read
Flag-Pattern:
Resource-Usage-Pattern:  'Records=' # → amount
Period:
Variable-Type:           ResourceByActivity
Threshold:               4                {standard deviations from mean}
```

This type of profile might be defined for individual users and files or classes thereof. An abnormality could signify an attempt to obtain sensitive data by inference and aggregation (e.g., by obtaining vast amounts of related data).

- *ReadFails, WriteFails, DeleteFails, CreateFails* -- event counters that

measure the number of access violations per day using the operational model. The structure of ReadFails is:

```
Variable-Name:          ReadFails
Action-Pattern:         'read'
Exception-Pattern:      'read-viol'
Resource-Usage-Pattern:
Period:                 day
Variable-Type:          SimpleCounter
Threshold:              2              {fixed bound}
```

This type of profile might be defined for individual users and files or classes thereof. Profiles for individual users and the class of all files could be useful for detecting users who persistently attempt to access unauthorized files. Profiles for individual files and the class of all users could be useful for detecting any unauthorized access to highly sensitive files (the threshold may be set to 1 in that case).

- *FileResourceExhaustion* -- event counter that measures the number failures caused by attempts to overflow the quota of available space using the operational model:

```
Variable-Name:          FileResourceExhaustion
Action-Pattern:         'write'
Exception-Pattern:      'resource-exhaust'
Resource-Usage-Pattern:
Period:                 day
Variable-Type:          SimpleCounter
Threshold:              2              {fixed bound}
```

This type of profile may be defined for individual users aggregated over all files. An abnormality might signify a covert channel, where the signaling process consumes all available disk space to signal a '1' bit.

Since the number of files in the system may be enormous (e.g., millions), it may be infeasible to monitor all files individually. It is, however, desirable to monitor accesses to the program files listed previously as well as to other data files that are security relevant or sensitive. Examples include:

password file
all files with authorization data
audit data
network host tables
address files, e.g., for electronic mail

4.2.5. Database-Access Profiles

Database accesses can be handled in much the same way as file accesses. In relational database systems, for example, every relation is stored as a separate file. Thus, accesses at the relation level are simply file accesses, though the operations are slightly different; in particular, we have 'retrieve', 'update', 'insert', and 'delete' for the records within a relation, and 'create' and 'delete' for a relation as a whole. Thus, database retrievals correspond to file reads, and database updates, inserts, and deletes correspond to file writes.

If auditing is performed at the relation (file) level, then exactly the same types of profiles used to monitor file activity can be used to monitor database activity. Indeed, a relation can simply be regarded as a special type of file. Moreover, this approach can be used with database systems that are not relational as well since all database systems store sets of records having common formats in files.

If auditing is desired at a lower level, say on individual records, the same basic principles apply as for files, but additional support is needed from the database system to write out individual audit records for all database records accessed. The object field for these records must uniquely identify the records, e.g., by relation name and record key. Because of the potentially huge volume of data generated by monitoring at the record level (a single retrieval could access a million records in order to compute an average value for the records), an intrusion-detection system operating at the record level may be infeasible[10].

For some systems, it may even suffice to monitor at the database level, without decomposing a database into its constituent files. Again, the same general principles apply as for files.

---

[10]This, of course, does not mean that access controls cannot be applied at the record or element level.

4.2.6. Profiles for Other Activities

Profiles of the type we have described can be defined for any type of activity that can be represented in the audit records. In particular, they can be defined for system-dependent or user-defined object types.

4.2.7. New Users and Objects

Introducing new users (and objects) into the target system potentially raises two problems. The first, which is caused by the lack of profile information about the user's behavior as well as by the user's own inexperience with the system, is generating an excessive number of anomaly records. This problem could be solved by ignoring anomalies for new users were it not for the second problem: failing to detect an intrusion by the new user. We would like a solution that minimizes false alarms without overlooking actual intrusions.

False alarms can be controlled by an appropriate choice of statistical model for the activities causing the alarms and by an appropriate choice of profiles. With the mean and standard deviation model, for example, the confidence intervals are initially large so that more diversity is tolerated while data are being collected about a user's behavior; the intervals then shrink as the number of observations increases. This reduces false alarms caused by an individual user profile, but does not protect the system against new users (or infrequent users) whose behavior is devious, or against users who establish unusual behavior from the beginning, as a cover. To deal with this problem, current activity can be compared with that in aggregate individual profiles or with the set of profiles for all users or all users in some group.

Although the operational model does not automatically adapt to an individual user (because it uses fixed thresholds to determine abnormality), the problem can be solved by using more lenient bounds with new users, and adjusting the bounds as the user gains experience.

4.3. Activity Rules

The activity rules that process audit data and update profiles depend on system-independent algorithms that encode different types of statistical metrics and models, and on parameters that are obtained from the components of a particular audit record and profile, namely the threshold for the anomaly test, the distribution of previous observations, the value of the current observation, and so forth. Similarly, the rules that generate periodic reports of anomalies depend on the statistical reporting methods (rules for reporting individual anomalies do not seem to require any statistical analysis). This section gives possible rules for managing the six variable types defined in Section 4.2.1, and gives examples of rules for anomaly records.

4.3.1. Audit-Record and Periodic-Activity-Update Rules

In Section 3.5, we defined a generic audit-record rule for type t that is triggered whenever a new audit record matches the patterns given in an activity profile and the Variable-Type component of the matching profile contains type t. The body of the rule is encoded in a procedure AuditProcess$t$, which was left unspecified. Similarly, we defined a generic periodic-activity-update rule for type t that is triggered whenever a period of length p ends and an activity profile contains p for its Period component and t for its Variable-Type component. The body of this rule is encoded in a procedure PeriodicProcess$t$, which was also left unspecified. We will now give possible definitions for these procedures for the six variable types.

First, consider simple event counters as defined in Figure 4-4:

```
          {Simple event counter with operational model}

AuditProcessSimpleCounter(Audit-Record, Profile)
  BEGIN
    NewSimpleCounter(Profile.Value);
    IF CheckSimpleCounter(Profile.Value, Profile.Threshold)
         THEN NewAnomaly('audit', Audit-Record.Time-stamp, Profile);
  END


PeriodicProcessSimpleCounter(Clock, Profile)
  BEGIN
    IF CheckSimpleCounter(Profile.Value, Profile.Threshold)
         THEN NewAnomaly('period', Clock, Profile);
    UpdateSimpleCounter(Profile.Value);   {reset counter}
  END
```

Note that every time an audit record is generated, the audit-record procedure checks the frequency of activity for the current time interval. In addition, the periodic procedure checks the frequency at the end of the time interval. Although checking only at the end of the time interval suffices for detecting abnormal behavior, there can be a delay between the time of intrusion and time of detection, especially if the time interval is long. By checking on each occurrence, we can detect a sudden flurry of activity as soon as it passes the threshold. Of course, it then becomes unnecessary to check at the end of the interval unless the value is compared against a lower bound to detect unusually low levels of activity.

Consider next an interval timer as defined in Figure 4-5. Here there is no need for a periodic procedure since the timer is not reset.

```
          {Interval timer with operational model}

AuditProcessIntervalTimer(Audit-Record, Profile)
  BEGIN
    NewIntervalTimer(Profile.Value, Audit-Record.Timestamp);
    IF CheckIntervalTimer(Profile.Value, Profile.Threshold)
         THEN NewAnomaly('audit', Audit-Record.Time-stamp, Variable);
  END
```

Procedures for the two types of event counters defined in Figures 4-6 and 4-7 are as follows:

```
        {Event counters with mean and standard deviation model}

AuditProcessEventCounter(Audit-Record, Profile)
  BEGIN
    NewEventCounter(Profile.Value);
    IF CheckEventCounter(Profile.Value, Profile.Threshold)
        THEN NewAnomaly('audit', Audit-Record.Time-stamp, Profile);
  END


PeriodicProcessEventCounter(Clock, Profile)
  BEGIN
    IF CheckEventCounter(Profile.Value, Profile.Threshold)
         THEN NewAnomaly('period', Clock, Profile);
    UpdateEventCounterDistrib(Profile.Value);
  END



  {Event counter by day and hour with mean and standard deviation model}

AuditProcessEventCounterByDayAndHour(Audit-Record, Profile)
  BEGIN
    d = day(Audit-Record.Time-stamp);
    h = hour(Audit-Record.Time-stamp);
    NewEventCounterByDayAndHour(Profile.Value, d, h);
    IF CheckEventCounterByDayAndHour(Profile.Value,
                                     Profile.Threshold, d, h)
         THEN NewAnomaly('audit', Audit-Record.Time-stamp, Profile);
  END


PeriodicProcessEventCounterByDayAndHour(Clock, Profile)
  BEGIN
    d = day(Clock);
    h = hour(Clock);
    IF CheckEventCounterByDayAndHour(Profile.Value,
                                     Profile.Threshold, d, h)
          THEN NewAnomaly('period', Clock, Profile);
    UpdateEventCounterByDayAndHourDistrib(Profile.Value, d, h);
  END
```

The functions 'day' and 'hour' convert a time-stamp into day-of-week and hour-of-day respectively.

Finally, we give procedures for the two types of resource measures defined in Figures 4-8 and 4-9. Since the type ResourceByActivity is not accumulated by period, it has no periodic-update rule. The parameter 'amount' in the rules refers to the matching

resource-usage value in the audit record (recall that the definition for a variable of these types includes a resource-usage pattern of the form "*name*=' # → amount').

```
            {Resource usage measured per activity}

AuditProcessResourceByActivity(Audit-Record, Profile)
  BEGIN
    NewResourceByActivity(Profile.Value, new);
    IF CheckResourceByActivity(Profile.Value, Profile.Threshold, amount)
         THEN NewAnomaly('audit', Audit-Record.Time-stamp, Profile);
  END



            {Resource usage measured over time}

AuditProcessResourceByTime(Audit-Record, Profile)
  BEGIN
    NewResourceByTime(Profile.Value, new);
    IF CheckResourceByTime(Profile.Value, Profile.Threshold, amount)
         THEN NewAnomaly('audit', Audit-Record.Time-stamp, Profile);
  END


PeriodicProcessResourceByTime(Clock, Profile)
  BEGIN
    IF CheckResourceByTime(Profile.Value, Profile.Threshold)
         THEN NewAnomaly('period', Clock, Profile);
    UpdateResourceByTimeDistrib(Profile.Value);
  END
```

4.3.2. Anomaly-Record and Periodic-Anomaly-Analysis Rules

Each anomaly-record rule specifies those anomaly records that may be related to a particular type of intrusion and should be brought to the immediate attention of the security officer. Ideally, the security officer should be notified of an abnormality only if there is a suspected intrusion. Unfortunately, we have very little knowledge about the exact relationship between certain types of abnormalities and intrusions (performing experiments to determine these relationships is an important objective of our proposed follow-on research).

In those cases where we do have experience, we can write rules that incorporate our knowledge. An example is with password failures, where the security officer should be notified immediately of a possible break-in attempt if the number of password failures

on the system during some interval of time is abnormal. This is expressed by the following rule, which has the form outlined in Section 3.5 (a more elaborate rule might also print the number of failures, whether they originated from one account or many, and the relevant audit record):

```
ANOMALY-RECORD RULE
    Condition: new Anomaly-Record
               Anomaly-Record.Profile.Variable-Name = 'PasswordFails'
               Anomaly-Record.Event = 'audit'

    Body: PrintAlert('Possible break-in: abnormal number of password failures',
                     Anomaly-record);
  END
```

Other abnormalities that are candidates for immediate notification include:

- Abnormal lapse since login (variable LastLogin in login profiles), which could indicate a successful break-in on a "dead" account.
- Highly abnormal login occurrence (variable LoginFrequency in login profiles) -- e.g., the user has never previously logged in late at night -- which could indicate masquerading.

Each periodic-anomaly-analysis rule analyzes some subset of the anomaly records generated during the time interval handled by the rule. It may produce a report summarizing the abnormalities found during the reporting period, without attempting to link the abnormalities to possible intrusions, or it may speculate about a possible intrusion. An example of a rule of this type is one that prints the total number of warnings generated during the time interval for each unique subject identified in the Subject field of a profile (grouping could also be over some other field such as Object or Action), but does not attempt to link the abnormalities to any particular type of intrusion:

```
PERIODIC-ANOMALY-ANALYSIS RULE
  Condition: Clock mod p = 0

  Body:
    Start = Clock - p;
    A = SELECT FROM Anomaly-Records WHERE Anomaly-Record.Time-stamp > Start;
    report = SELECT COUNT() FROM A
       GROUP BY Anomaly-Record.Profile.Subject-Pattern;
    Print(report);
END
```

The above rule uses the "GROUP BY" feature of relational programming so that a separate total is given out for each user (or group) that appears in the anomaly records.

Rules that process anomaly records might produce summary tables of statistics broken down by one or more categories or graphs of abnormalities. They might compute statistical functions over anomalies in order to link them to possible intrusions. Thus far, we do not have enough experience with on-line intrusion detection to know exactly how to relate abnormalities to intrusions or what reports will be the most useful. Some experience with an actual system is needed first to determine useful rules for processing the anomaly records.

5. System Design

This section outlines a possible design strategy for IDES and its knowledge base.

5.1. System Configuration

Figure 5-1 shows a system diagram for IDES, where IDES is implemented on a processor that is physically separate from that of the system monitored. The connection between IDES and the system monitored is used only for downloading audit records to IDES. The only way of interacting with IDES is by direct connection to the IDES processor, and this path can be physically restricted to the security officer.

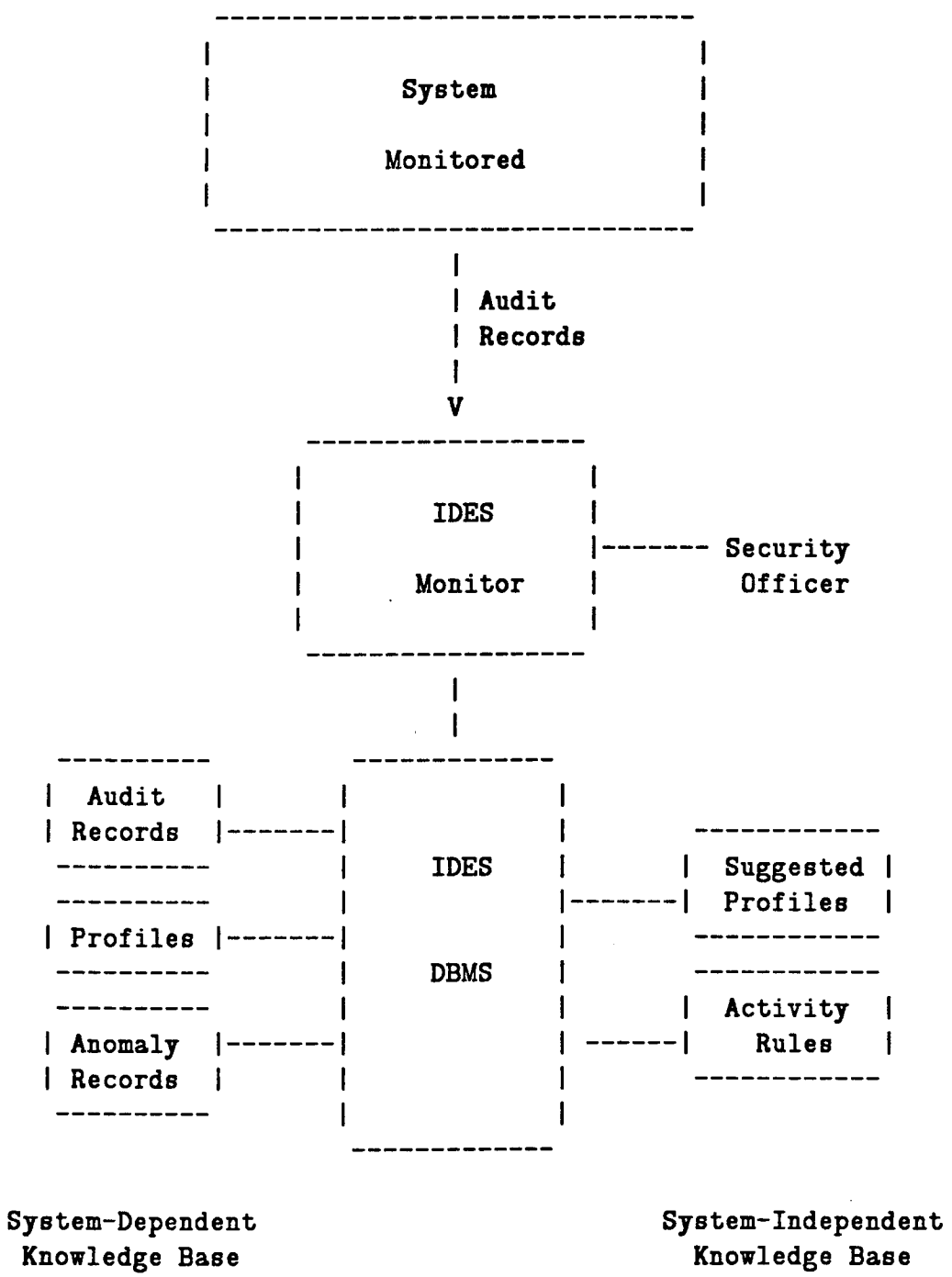
Implementing IDES on a separate processor has both performance and security advantages. The performance advantage is that IDES will not noticeably degrade the response time of the system monitored or otherwise affect its behavior (there may be some delay associated with the communication link). The security advantage is that IDES is protected from the target system. In particular, a user on the target system cannot gain access to IDES or its knowledge base in order to tamper with profiles, anomaly records, rules, etc. Thus, whatever flaws may exist in the system monitored do not endanger IDES and its knowledge base.

An intruder might, however, attempt to subvert IDES by tampering with the audit records before they are transmitted from the target system to IDES -- e.g., deleting all audit records associated with his session. There are several strategies that can be used to protect against such tampering:

- Place sequence numbers in the audit records to detect deletions.
- Place cryptographic checksums in all audit records to detect tampering. Because the key is shared by the target system and IDES, this is effective only if the key is adequately protected on the target system.
- Monitor access to the audit procedures.

Of course, if an intruder successfully penetrates the system, he may be able to turn off all auditing and completely escape detection while performing considerable damage. A goal of IDES is to detect the penetration before that happens.

Figure 5-1: System Configuration.



IDES itself has two main components: a monitor, which interfaces with the main system and the security officer, and a database management system, which manages the IDES knowledge base.

5.2. IDES Monitor

The IDES monitor accepts audit records from the target system and interfaces with the security officer. Audit data are handled by a demon process that listens for data on the link connecting IDES to the target system. When an audit record is received, it is passed to the database management system for processing.

The interface to the security officer is used mainly for defining profile templates (although some templates will be built-in, the security officer may wish to modify these or define others) and reporting anomalies to the security officer. In addition, it gives the security officer a means of querying the IDES database for information about the current status of profiles or anomalies. The interface may also allow the officer to tune IDES to the target system by adding or modifying metrics, profile templates, and rules. For example, built-in measures that are found to be ineffective for a particular system might be modified or deleted.

The monitor is also responsible for initiating all periodic processing of activity profiles and anomaly records. Periodic-activity-update rules are fired before periodic-anomaly-analysis rules since the former can generate additional anomaly records that should be included in the analysis.

The IDES monitor should be system-independent, with all system-dependent information stored in the IDES database. One area where there is great diversity among different systems is in audit record formats. This diversity could be resolved by providing a system-independent audit record interface to IDES. The audit records of a particular system would then pass through a filter that puts them in the standard format before being transmitted to IDES.

5.3. IDES Knowledge Base and Database Management System

The IDES database management system (IDBMS) manages both the system-dependent and system-independent knowledge bases. The data itself could be stored and processed as relations, possibly using an off-the-shelf relational database management system.

The system-dependent knowledge base can be described in terms of three main relations:

- *Audit-Records* -- for all audit records. Because this relation will be enormous (there might be a million records per week), it should be physically represented by a multilevel store, where records are migrated from the main store (e.g., high-speed disk) to secondary store (e.g., tape or optical disk).
- *Profiles* -- for all activity profiles and profile templates.
- *Anomaly-Records* -- for all anomaly records.

Note that the above relations are not necessarily in 'normal form'. An implementation may decompose these relations into normal form to preclude certain kinds of update, insert, and delete 'anomalies' on the data and to minimize the storage requirements -- we do not consider how this should be done here.

The system-independent knowledge base could be defined by relations that contain suggested profile templates and information about the statistical metrics, models, and activity rules built into IDES. The rules themselves may be encoded as update and retrieval queries on the relations. For example, consider the audit-record rules, which are triggered by a new audit record, a match between the record and a profile, and a match between the type in the profile and the type implemented by the rule. When a new audit record is received, a query on the Profiles relation can retrieve all activity profiles and template profiles matching the audit record, generating new activity profiles for templates that do not have matching activity profiles. For each activity profile in the set, a query that implements the AuditProcess rule for that type is then invoked. Periodic-activity-update rules can be similarly handled. Periodic-anomaly-analysis rules, which may compute statistics over past anomaly records, can be implemented as queries

on the Anomaly-Record relation. Note that, unlike some expert systems, it is important in IDES that all rules whose conditions are satisfied be fired.

Commercial relational database systems may not have all of the capabilities desired for IDES, particularly for pattern matching and implementing statistical models that are more complex than those based on mean and standard deviation. These capabilities may be provided through a higher-level language that interfaces to the database query language (via compilation/interpretation or procedure calls). Although we do not attempt to define such a language here, we do note that Prolog and LISP, which are popular for AI applications, are not appropriate for IDES. Neither supports statistical computing. Although Prolog provides matching (in the form of resolution) its matching capability does not seem to be well suited to IDES (recursive matching with backup, for example, may not be needed).

6. Research Questions

Although we believe the approach outlined in this report is powerful and feasible, further research is needed to address several questions:

- *Soundness of Approach* -- Does the approach actually detect intrusions? Is it possible to distinguish anomalies related to intrusions from those related to other factors? It is not possible to answer these questions without experimentally trying the approach and measuring the correlation between anomalies and actual (or simulated) intrusions.

- *Completeness of Approach* -- Does the approach detect most, if not all, intrusions, or is a significant proportion of intrusions undetectable by this method? We expect that the approach can detect most intrusions, though subtle forms of intrusion that use low-level features of the target system that are not monitored (because they would produce too much data) may escape notice. For example, because it is not practical to monitor individual page faults, a program that leaks data covertly by controlling page faults would not be detected -- at least by its page-fault activity.

  It is important, however, to distinguish between detecting a low-level action that exploits a particular flaw in a system and detecting the related intrusion. For example, if the program that acquires the sensitive data covertly routes the data to an abnormal location, or the user acquiring the data logs into the system during unusual hours or from an abnormal location, the intrusion may be detected from the aberrant activity. As another example, consider an operating system penetration based on trying supervisor calls with slight variants of the calling parameters until arguments are found that allow the user to run his own programs in supervisor mode. Detecting the actual penetration would be impossible without monitoring all supervisor calls, which is generally not practical. The intrusion, however, may be detected once the penetration succeeds if the intruder begins to access objects he could not access before. Thus, the important question is not whether IDES can detect a particular low-level action that exploits a system flaw, but whether intrusions are manifest by activity that can be practically monitored.

  In order to determine the ability of IDES to detect intrusions, it will be necessary to experimentally try different methods of intrusion and observe whether, and how quickly, IDES detects the intrusions. A significant challenge for IDES will be responding to someone attempting live penetration testing.

- *Timeliness of Approach* -- Can IDES detect intrusions before significant damage is done? The answer to this question depends on whether the actions that trigger detection sufficiently precede those that cause damage that the

security officer can be notified of the possible intrusion and take appropriate action. This is the main reason why IDES monitors all activity for aberrant usage, and not just security-related activity or actions that could cause potential damage (e.g., file deletes).

IDES can never replace the security controls on the target system, which aim to prevent intrusions and ensuing damage. Its operating mode is purely detective. A swift one-shot penetration that exploits a system flaw and then destroys all files with a single 'delete all' command may be able to perform significant damage before any action can be taken. It is doubtful that one can do much to protect against this threat beyond what can be provided by the security controls on the target system.

- *Choice of Metrics, Statistical Models, and Profiles* -- What metrics, models, and profiles provide the best discriminating power? Which are cost-effective? What are the relationships between certain types of anomalies and different methods of intrusion? In Section 4 we gave suggestions for measures and rules, but these were based on intuition and our desire to limit the examples to a few simple measures. We need more information from security officers who have observed the effect of actual intrusions on system activity, as well as experimental data showing correlations between actual intrusions and different statistical measures.

- *System Design* -- How should the IDES software and database be structured? What should the user (security officer) interface to IDES look like? What should be the standard format for audit records? What language should be used for defining profile templates, pattern matching, and writing rules (which require statistical computations)? In Section 5 we sketched a design based on a relational database system, but left the details for future work.

- *Social Implications* -- How will IDES affect the user community it monitors? Will it deter intrusions? Will the users feel their data are better protected? Will it be regarded as a step towards 'big brother' watching all of our activity and invading our privacy? Will its capabilities be misused to that end? It is not enough to consider technical questions; the social aspects could significantly affect the ultimate successes or failure of IDES.

7. Conclusions and Future Work

We have developed a model for an intrusion-detection expert system called IDES, showed how the model can be applied to real systems, and outlined a system-independent design that allows the IDES software to be independent of the system monitored. We believe that the approach is feasible and capable of detecting a wide range of intrusions related to attempted break-ins, masquerading (successful break-ins), system penetrations, Trojan horses, viruses, leakage and other abuses by legitimate users, and certain covert channels. Moreover, it can detect intrusions without knowing about the flaws in the target system that allowed the intrusion to take place, and without necessarily observing the particular action that exploits the flaw. Although IDES may be unable to detect all intrusions, we believe that it can detect most intrusions and thereby significantly enhance the security of a target system.

We recommend development of a rapid prototype of IDES to test the model and approach experimentally. The prototype would provide a testbed for investigating different metrics, statistical models, and rules, and for measuring the correlation between simulated intrusions and observed anomalies. It would also provide a means of refining the design of IDES, including the IDES knowledge base and user interface. If the prototype effort demonstrates the feasibility and effectiveness of the approach, we recommend development of a production IDES that would be capable of monitoring a wide variety of systems and applications.

The approach outlined in this report may apply to areas other than security -- for example, it may be useful for detecting unreliable system states (e.g., a system that is beginning to crash) and dangerous situations involving human safety (e.g., failures in control systems for traffic, ships, aircraft, and missiles). If IDES is successful in detecting security breaches, then investigating its application to these other areas would be worthwhile.

I. Sample Cases of Intrusion

- San Francisco Public Defender files were inadvertently made accessible to police and prosecutors. As many as 1000 cases could have been compromised as a result, although the lack of adequate audit trails makes it impossible to know what information might actually have been read, and by whom.

- Someone gained access to the password file for British Telecom's Prestel Information Service and demonstrated this to a reporter for the London Daily Mail -- by reading Prince Philip's mailbox and altering a financial market database.

- In a clever 1984 April-Fool's-Day hoax, someone faked a message from Chernenko@MOSKVAX and modified the system tables to permit the receipt of return mail sent to the originating (but in reality nonexistent) host.

- Someone accessed the TRW on-line credit information bureau (which is widely available), and then obtained a credit card number for a Newsweek reporter (Richard Sandza) who had written "anti-hacker" articles. $1100 in unauthorized charges resulted, apparently in retribution for the article.

- High-school students (the "Milwaukee 414s") found it relatively trivial to break in to a large number of computer systems using the ARPANET. Although they seem to have been mostly snooping, they also did some damage.

- Bloodstock Research Inc. (in Kentucky), which maintains genealogical and other data on thoroughbred race horses, had its system logged into via dial-up and its database compromised.

- A Santa Clara County jail inmate managed to log in to the prison computer database system and alter his release date.

- Cal Tech students remotely subverted the scoreboard controls during the 1984 Rose Bowl, and altered the display (e.g., changing the teams to "Cal Tech vs. MIT").

Further background on these cases can be found in recent issues of the ACM SIGSOFT Software Engineering Notes. Over a hundred additional cases of computer-related security problems are documented in Norman [4].

II. Security Flaws in Computer Systems

The following is a list of a few known flaws in different operating systems. For each flaw, we suggest how an intrusion that exploits the flaw might be detected by IDES.

- TENEX's lack of hang-up detection leaves a dialed-up process logged in after a line break, accessible to whomever next happens to acquire that dial-up port -- detect masquerader by abnormal usage of account.

- An earlier Cyber system had a command that complained if a would-be new password was already in use by another user -- detect attempts to find the account associated with the password by abnormal password failures; detect successful break-in by abnormal usage of account.

- UNIX passwords were subject to dictionary attacks despite encrypted passwords [1] -- detect attempted break-in by unusually high number of password failures; detect successful break-in by abnormal usage of an account.

- UNIX Version 6 had an accidental universal password that worked for all users -- detect masquerading by abnormal usage of account.

- UNIX 'root' capture is possible, even without being logged in -- detect by abnormal activity for 'root' account.

- UNIX Version 6 login could be broken by successive quits during login -- detect successful break-in by abnormal usage of account.

- A Trojan horse was implanted in the UNIX C compiler, awaiting the next recompilation of the UNIX login procedure, upon which a trap door was installed that allowed login into any account using a special password [7] -- detect unauthorized use of account by abnormal usage.

- Mounted file systems in UNIX (for example) bypass protection checks -- detect by abnormal file accesses.

- Serious flaws in the security appliques RACF and ACF-2 and in their underlying operating system MVS result in easy penetrability [5] -- detect successful penetration by change in files and commands accessible to intruder.

- UNIX restricted shells are trivially breakable, in a variety of ways, allowing one to acquire Superuser status -- detect penetration by change in files and commands accessible to intruder.

- UNIX Superuser status and TOPS-20 Wheel status are vastly too powerful, allowing considerable damage to be done by a successful penetration -- this may facilitate intrusion detection because the penetrator's activity can change dramatically once privileged status is acquired.

- TOPS-20 retrieval of an archived file that had been protected brings it back unprotected -- detect by abnormal file access.

- In UNIVAC 1100 under Exec-8, a chain of 8 indirect words resulted in memory protection being bypassed for the last access; this could be used to penetrate the system (by overwriting system tables) -- detect by abnormal command and file usage by penetrator.

A summary of characteristic flaws in computer systems is given in Neumann [2] and includes weaknesses in systems and in programming languages such as improper encapsulation, naming problems, hidden side-effects, improper validation, interruptible "atomic" actions, improper sequencing, etc.

References

1. R. Morris, K. Thompson. "UNIX Password Security: A Case History". *Comm. ACM 22*, 11 (November 1979), 594-597.

2. Neumann, P. G. "Computer Security Evaluation". *Proc. NCC 47* (1978), 1087-1095.

3. P.G. Neumann. Audit Trail Analysis and Usage Data Collection and Processing, Part One. SRI International, Project 5910, January, 1985.

4. A.R.D. Norman. *Computer Insecurity.* Chapman and Hall Ltd., New York and London, 1983.

5. Ronald Paans, Guus Bonnes. Surreptitious Security Violation in the MVS Operating System. Delft University of Technology, Department of Electrical Engineering, POBox 5031, 2600 GA Delft, Netherlands, 1984.

6. *System Management Facilities.* BC28-0706-1 edition, IBM Corp., 1977.

7. K. Thompson. "Reflections on Trusting Trust (1983 Turing Award Lecture)". *Comm. ACM 27*, 8 (August 1984), 761-763.

8. *UNIX Programmer's Manual.* 4.2 Berkeley Software Distribution edition, Dept. EECS, Univ. of Calif., Berkeley, 1983.