AN INTRUSION-DETECTION MODEL

Dorothy E. Denning

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025.

A model of a real-time intrusion-detection expert system capable of detecting break-ins, penetrations, and other forms of computer abuse is described. The model is based on the hypothesis that security violations can be detected by monitoring a system's audit records for abnormal patterns of system usage. The model includes profiles for representing the behavior of subjects with respect to objects in terms of metrics and statistical models, and rules for acquiring knowledge about this behavior from audit records and for detecting anomalous behavior. The model is independent of any particular system, application environment, system vulnerability, or type of intrusion, thereby providing a framework for a general-purpose intrusion-detection expert system.

1. Introduction

This paper describes a model for a real-time intrusion-detection expert system that aims to detect a wide range of security violations ranging from attempted break-ins by outsiders to system penetrations and abuses by insiders. 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.

The model 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:

- *Attempted break-in* -- Someone attempting to break into a system might generate an abnormally high rate of password failures with respect to a single account or the system as a whole.

- *Masquerading or successful break-in* -- Someone logging into a system through an unauthorized account and password might have a different login time, location, or connection type from that of the account's legitimate user. In addition, the penetrator's behavior may differ considerably from that of the legitimate user; in particular, he might spend most of his time browsing through directories and executing system status commands, whereas the legitimate user might concentrate on editing 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* -- A user attempting to penetrate the security mechanisms in the operating system might 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* -- A user trying to leak sensitive documents might log into the system at unusual times or route data to remote printers not normally used.

- *Inference by legitimate user* -- A user attempting to obtain unauthorized data from a database through aggregation and inference might retrieve more records than usual.

- *Trojan horse* -- The behavior of a Trojan horse planted in or substituted for a program may differ from the legitimate program in terms of its CPU time or I/O activity.

- *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.

- *Denial-of-Service* -- An intruder able to monopolize a resource (e.g., network) might have abnormally high activity with respect to the resource, while activity for all other users is abnormally low.

Of course, 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. An important objective of our current research is to determine what activities and statistical measures provide the best discriminating power; that is, have a high rate of detection and a low rate of false alarms.

2. Overview of Model

The model is independent of any particular system, application environment, system vulnerability, or type of intrusion, thereby providing a framework for a general-purpose intrusion-detection expert system, which we have called IDES. A more detailed description of the design and application of IDES is given in our final report[1].

The model has six main components:

- *Subjects* -- initiators of activity on a target system -- normally users.

- *Objects* -- resources managed by the system -- files, commands, devices, etc..

- *Audit records* -- generated by the target system in response to actions performed or attempted by subjects on objects -- user login, command execution, file access, etc.

- *Profiles* -- structures that characterize the behavior of subjects with respect to objects in terms of statistical metrics and models of observed activity. Profiles are automatically generated and initialized from templates.

- *Anomaly records* -- generated when abnormal behavior is detected.

- *Activity rules* -- actions taken when some condition is satisfied, which update profiles, detect abnormal behavior, relate anomalies to suspected intrusions, and produce reports.

The model can be regarded as a rule-based pattern matching system. When an audit record is generated, it is matched against the profiles. Type information in the matching profiles then determines what rules to apply to update the profiles, check for abnormal behavior, and report anomalies detected. The security officer assists in establishing profile templates for the activities to monitor, but the rules and profile structures are largely system-independent.

The basic idea is to monitor the standard operations on a target system: logins, command and program executions, file and device accesses, etc., looking only for deviations in usage. The model does not contain any special features for dealing with complex actions that exploit a known or suspected security flaw in the target 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 would be unable to cope with intrusions that exploit deficiencies that are not suspected or with personnel-related vulnerabilities. By detecting the intrusion, however, the security officer may be better able to locate vulnerabilities.

The remainder of this paper describes the components of the model in more detail.

3. Subjects and Objects

Subjects are the initiators of actions in the target 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.

4. 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. This should be the actual exception condition raised by the system, not just the apparent exception condition returned to the subject.
- *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* -- unique time/date stamp identifying when the action took place.

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.

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. 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. Our intrusion-detection model 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, since activity is observed without regard for authorization, there is an implicit assumption that the access controls in the system permitted an action to occur. The task of intrusion detection is to determine whether activity is unusual enough to suspect an intrusion. Every statistical measure used for this purpose is computed from audit records associated with one or more cells in the matrix.

Most operations on a system involve multiple objects. For example, file copying involves the copy program, the original file, and the copy. Compiling involves the compiler, a source program file, an object program file, and possibly intermediate files and additional source files referenced through "include" statements. Sending an electronic mail message involves the mail program, possibly multiple destinations in the "To" and "cc" fields, and possibly "include" files.

Our model decomposes all activity into single-object actions so that each audit record references only one object. File copying, for example, 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. The following illustrates the audit records generated in response to a command

```
COPY GAME.EXE TO <Library>GAME.EXE
```

issued by user Smith to copy an executable GAME file into the <Library> directory; the copy is aborted because Smith does not have write permission to <Library>:

```
(Smith, execute, <Library>COPY.EXE, 0,
      CPU=00002, 11058521678)
(Smith, read,    <Smith>GAME.EXE,   0,
       RECORDS=0, 11058521679)
(Smith, write,   <Library>GAME.EXE, write-viol,
       RECORDS=0, 11058521680)
```

Decomposing complex actions has three advantages: First, since objects are the protectable entities of a system, the decomposition is consistent with the protection mechanisms of systems. Thus, 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.

The target system is responsible for auditing and for transmitting audit records to the intrusion-detection system for analysis (it may also keep an independent audit trail). 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 both successful and unsuccessful attempts to perform the activity, even if the action should abort (e.g., because of a protection violation) 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)[2], for example, audit only the completion of these activities.

Although the auditing mechanisms of existing systems approximate the model, they are typically deficient in terms of the activities monitored and record structures generated. For example, Berkeley 4.2 UNIX[3] monitors command usage but not file accesses or file protection violations. Some systems do not record all login failures. Programs, including system programs, invoked below the command level are not explicitly monitored (their activity is included in that for the main program). The level at which auditing should take place, however, is unclear, since too much auditing could severely degrade performance on the target system or overload the intrusion-detection system.

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. 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 the values. A uniform record format with self-identifying data would be preferable so that the intrusion-detection software can 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 format.

5. Profiles

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.). Observations (sample points) x_i of x obtained from the audit records 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.

Before describing the structure, generation, and application of profiles, we shall first discuss statistical metrics and models.

5.1. Metrics

We define 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 are number of logins during an hour, number of times some command is executed during a login session, and number of password failures during a minute.

- *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. An example is the length of time between successive logins into an account.

- *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 and total amount of CPU time consumed by some program during a single execution. Note that a resource measure in our intrusion-

detection model 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. Thus, whereas event counters and interval timers 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.

5.2. 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.

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:

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

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

$$mean = sum/n$$

$$stdev = sqrt(\frac{sumsquares}{(n-1)} - mean^2) .$$

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. Note that 0 (or null) occurrences should be included so as not to bias the data.

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, what is considered to be normal for one user can be considerably different from another.

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.

5.3. 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.
- *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 the model leaves unspecified the exact format for patterns, we have identified the following SNOBOL-like constructs as being useful:

'*string*'	string of characters
*	wild card matching any string
#	match any numeric string.
IN(*list*)	match any string in *list*.
$p \rightarrow name$	the string matched by p is associated with *name*
$p1$ $p2$	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 to User
'<Library>*' -- match files in <Library> directory
IN(Special-Files) -- match files in Special-Files
'CPU=' # → Amount --  match string 'CPU=' followed
                by integer; assign integer to Amount
```

The following is a sample profile for measuring the quantity of output to user Smith's terminal on a session basis. The variable type ResourceByActivity denotes a resource measure using the mean and standard deviation model.

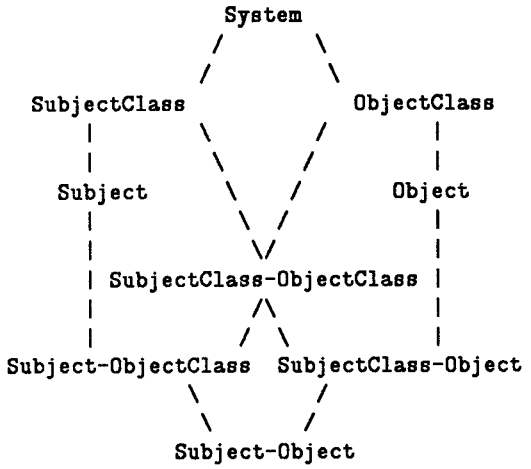
```
Variable-Name:           SessionOutput
Action-Pattern:          'logout'
Exception-Pattern:       0
Resource-Usage-Pattern:  'SessionOutput=' # → amount
Period:
Variable-Type:           ResourceByActivity
Threshold:               4
Subject-Pattern:         'Smith'
Object-Pattern:          *
Value:                   record of ...
```

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

5.4. Profiles for Classes

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 5-1. 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:

Figure 5-1: Hierarchy of Subjects and Objects.



- *Subject - Object*: actions performed by single subject on single object -- e.g., user Smith - file Foo.

- *Subject - Object Class*: actions performed by single subject aggregated over all objects in the class. The class of objects might be represented as a pattern match on a subfield of the Object field that specifies the object's type (class), as a pattern match directly on the object's name (e.g., the pattern '*.EXE' for all executable files), or as a pattern match that tests whether the object is in some list (e.g., "IN(hit-list)").

- *Subject Class - Object*: actions performed on single object aggregated over all subjects in the class -- e.g., privileged users - directory file <Library>, nonprivileged users - directory file <Library>.

- *Subject Class - Object Class*: actions aggregated over all subjects in the class and objects in the class -- privileged users - system files, nonprivileged users - system files.

- *Subject*: actions performed by single subject aggregated over all objects -- e.g., user session activity.

- *Object*: actions performed on a single object aggregated over all subjects -- e.g., password file activity.

- *Subject Class*: actions aggregated over all subjects in the class -- e.g., privileged user activity, nonprivileged user activity.

- *Object Class*: actions aggregated over all objects in the class -- e.g., executable file activity.

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

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.

5.5. 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 creates all profiles.

2. Automatic explicit create -- All profiles for a new user or object are generated in response to a 'create' record in the audit trail.

3. First use -- A profile is automatically generated 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.

The IDES model 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'
'Jones'	'Accounts'	'Jones'	'Accounts'
'Smith'	'Foo'	no match, so no profile	

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>Let'	no new profile generated	
'Jones'	'<Jones>Foo'	'Jones'	'<Jones>*'
'Smith'	'<Jones>Foo'	no match so no profile	

5.6. 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.

5.7. Possible Profiles

We shall now describe candidate profiles for measuring login and session activity, command and program usage, and file accesses. For each profile, we suggest a metric and statistical model for measuring the activity. More complete profile specifications are given in the IDES final report[1].

5.7.1. 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. The following is a list of possible profiles:

- *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.) 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. Login profles might be defined for individual users (and user groups) but classes of locations -- either all locations taken together or aggregated by type of location or connection.

- *LocationFrequency* -- event counter that measures the frequency of login at different locations using the mean and standard deviation model. 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. 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. 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 using the mean and standard deviation model. 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). 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) using the mean and standard deviation model. These profiles may be useful for detecting masqueraders.

- *PasswordFails* -- event counter that measures password failures at login using the operational model. 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. 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. This type of profile might be defined for individual users, but aggregates of locations 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 privileged terminals.

5.7.2. Command or Program Execution

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, or by some other property.

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

- *ProgamCPU, ProgramIO, etc.* -- resource measures per execution of a program using the mean and standard deviation model. 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.

- *ExecutionDenied* -- event counter for number of attempts to execute an unauthorized program during a day using the operational

model. 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. 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).

5.7.3. File-Access Activity

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', 'delete', or 'append'. 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.

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. 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. 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. 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. 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.

6. Anomaly Records

Through its activity rules (next Section), 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 have unique 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).

7. 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.

7.1. 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$t$ (see the IDES final report[1] for example procedures). 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
```

7.2. Periodic-Activity-Update Rules

This type of rule, which is also parameterized by the type t of statistical measure, is triggered whenever the clock implies 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
```

7.3. 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
```

Unfortunately, we have very little knowledge about the exact relationship between certain types of abnormalities and intrusions. 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. Other abnormalities that are candidates for immediate notification include an abnormal lapse since last login or an abnormal login time or place -- e.g., the user has never previously logged in late at night -- both of which could indicate masquerading.

7.4. 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. 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 what reports will be the most useful.

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.

8. Conclusions

We believe the IDES model provides a sound basis for developing a powerful real-time intrusion detection 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, the model allows intrusions to be detected 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.

There are several open 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?

- *Completeness of Approach* -- Does the approach detect most, if not all, intrusions, or is a significant proportion of intrusions undetectable by this method?

- *Timeliness of Approach* -- Can we detect most intrusions before significant damage is done?

- *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?

- *System Design* -- How should a system based on the model be designed and implemented?

- *Feedback* -- What effect should detection of an intrusion have on the target system? Should IDES automatically direct the system to take certain actions?

- *Social Implications* -- How will an intrusion-detection system 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'? Will its capabilities be misused to that end?

Although we believe that the approach can detect most intrusions, it may be possible for a person to escape detection through gradual modifications of behavior or 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). 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. As an 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 we can detect a particular low-level action that exploits a system flaw, but whether intrusions are manifest by activity that can be practically monitored.

Even if we can detect most intrusions, we do not propose to replace any security controls with IDES. Doing so could leave the target system vulnerable to such attacks as a swift one-shot penetration that exploits a system flaw and then destroys all files with a single 'delete all' command. Our concept for IDES is an extra layer of protection that enhances the overall security of the target system.

Acknowledgments

I especially thank Peter Denning for stimulating and helpful discussions on statistical modeling while hiking through the redwoods, and Peter Neumann for working with me on the IDES project and helping me clarify my ideas. I also thank Hal Javitz and Karl Levitt for many useful comments and suggestions. Finally, I wish to acknowledge the Space and Naval Warfare Command

(SPAWAR) for funding the research under contract 83F830100, and the NSF for their support under grant MCS-8313650.

References

1. Denning, D. E. and Neumann, P. G., "Requirements and Model for IDES -- a Real-Time Intrusion Detection System", Tech. report, Computer Science Lab, SRI International, 1985.

2. IBM Corp., *System Management Facilities,* BC28-0706-1 ed., 1977.

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