NIDX - AN EXPERT SYSTEM FOR REAL-TIME NETWORK INTRUSION DETECTION

David S. Bauer
Bell Communications Research, Inc.
444 Hoes Lane (RRC 1H-226)
Piscataway, NJ 08854

Michael E. Koblentz
Bell Communications Research, Inc.
444 Hoes Lane (RRC 1J-221)
Piscataway, NJ 08854

Abstract

A knowledge-based prototype Network Intrusion Detection Expert System (NIDX) for the UNIX® System V environment is described. NIDX combines knowledge describing the target system, history profiles of users' past activities, and intrusion detection heuristics forming a knowledge-based system capable of detecting specific violations that occur on the target system. Intrusions are detected by classifying user activity from a real-time audit trail of UNIX system calls then, using system-specific knowledge and heuristics about typical intrusions and attack techniques, determining whether or not the activity is an intrusion. This paper describes the NIDX knowledge base, UNIX system audit trail mechanism and history profiles, and demonstrates the knowledge-based intrusion detection process.

1. INTRODUCTION

Corporate network security is rapidly increasing in importance to the Bellcore Client Companies (BCCs) due to the proliferation of distributed data and the introduction of applications of new technology and applications that allow customers and technicians remote access to network testing and control functions. Potential threats include the unauthorized browsing, modifying, deleting of stored information, the theft of data via high speed, fiber optic communications lines and the theft or malicious disruption of network services via control of network elements and network management systems through which these services are supported.

This paper describes the Network Intrusion Detection Expert System (NIDX) which is being prototyped by Bell Communications Research as part of our effort to address BCC network security issues. NIDX is a knowledge-based system that examines a real-time, detailed audit trail of user's actions on a computer system with the purpose of detecting security violations. We consider intrusion detection, or real-time audit trail analysis, to be one component of a more comprehensive security strategy. Other components are essentially preventive in nature, including data encryption, distributed authentication, secure operating systems, physical security and, most importantly, the definition and enforcement of personnel security policies.

NIDX combines knowledge describing the target system and intrusion detection heuristics to form a knowledge-based system capable of detecting specific violations that are potential threats to the target system. Initially, NIDX targets the UNIX® System V environment, but its concepts may later be applied to other operating systems as well.

The next section reviews previous work and introduces the knowledge-based approach to intrusion detection. Our knowledge-based model applied to the UNIX System V environment is presented in section 3. Sections 4-6 then present the NIDX architecture, knowledge base, audit trail, usage profiles and the intrusion detection process. The concluding section points to some issues for further investigation.

2. PREVIOUS WORK

The intrusion detection model is based on the premise that either in the attempt to break into a system, or once having broken into a system, an intruder will exhibit some form of abnormal behavior when attempting to access and control the system's resources. This abnormal behavior can then be detected by real-time analysis of an activity audit trail allowing appropriate alerting and recommendations for action to the system security administrator. This basic idea has been previously expressed by Denning, et al, [1, 2] who defined an Intrusion Detection Expert System (IDES) model. The IDES model is a general purpose, system-independent

* UNIX is a registered trademark of AT&T

model whose major components are:

- Subjects - entities that initiate some action, e.g. terminal users user programs, or system programs.
- Objects - entities acted upon by subjects, e.g. files, memory locations, or physical devices.
- Audit Records - logs of actions taken or attempted by subjects on objects.
- Profiles - structures that express the observed normal relationships between subjects and objects, e.g. the access frequency for a particular user and a particular file.
- Anomaly Records - structure generated when a subject's action on an object is abnormal with respect to the corresponding profile.
- Activity Rules - describe actions to be taken when some condition is satisfied, e.g. update a profile based on new observed behavior in the audit records, or send a report to the security administrator based on analysis of anomaly records.

In the IDES model, intrusions are detected using the following approach: first, profiles are initialized and updated based on observed behavior of normal users, then monitor events on the system ("actions") looking for anomalous behavior. Anomalous behavior is discovered by comparing a users' current actions to their profiled history and determining by statistical methods that a user logged in as user u is exhibiting behavior not consistent with u's usual behavior. For example, if a user performs actions (e.g. file removals) the number of which deviates substantially from the mean of that activity as shown by his/her profile, then that behavior is deemed anomalous. Once having detected such behavior, the system takes some appropriate action which may involve a report to the administrator.

The IDES model is intentionally system-independent to allow for applicability to a wide range of operating system environments. Although system-independence is the ideal objective, in practice this approach may be self limiting because *intruders* will exploit knowledge and attack techniques specific to the system under attack. Also, the technique of comparing users' past behavior to their current behavior to detect violations has limitations. For example, before violations can be flagged with any confidence, a substantial history profile must be developed for each user, thereby making it difficult to detect intrusions based on new user accounts. Finally, the statistical inference method described above may require many discrete events to exceed the thresholds. This makes it difficult to detect intruders in real-time and contain them before significant damage is done. In the next section we describe an alternative approach that addresses these issues using a knowledge-based, system-dependent model.

3. NIDX INTRUSION DETECTION MODEL

The NIDX model is based on the IDES system-independent model, but is extended to include target system dependent knowledge. The system dependent knowledge includes facts describing the target system and heuristics embodied in rules that detect particular violations from the target system audit trail. Although this model was developed for the UNIX system and some amount of system dependent information is necessary for accurate and fast intrusion detection, many of the concepts of the model should be applicable to other operating systems. The components of the UNIX system model are:

- Subjects - in the UNIX model, subjects are always users. Programs are merely the vehicle by which users interact with objects.
- Objects - entities acted upon by subjects (users), e.g. files, programs, devices, message queues, and directories.
- Intrusions - the set of intrusions that a UNIX system intrusion detection system discovers. For example, unauthorized reading, modification, or theft of information either internally or externally initiated.
- Suspicious Events - events that individually are not intrusions but that may be an indication of an impending security violation. For example, a user who logs in late at night who has no history of such activity.
- System Knowledge Base - contains knowledge about the system being monitored and heuristics for detecting intrusions.
- Audit Records - the UNIX system accounting software does not provide a detailed, reliable audit trail sufficient for intrusion detection. An audit trail of system calls is required for successful intrusion detection.
- Anomaly Records - structures generated when a subject's action on an object indicates a security violation.
- Profiles - each user has a set of permanent profiles that maintain a history of their normal interaction with the system. They are updated at the end of each login session to reflect that day's computer use. Profiles reflect each users relationship to objects on the system, for example, user x to objects owned by another user y.

- Rules - used to update session profiles from audit records, detect suspicious events from audit records, detect anomalies which are indications of intrusions, and reason about anomaly records to detect intrusions.

4. NIDX IMPLEMENTATION

This section describes the implementation of NIDX for intrusion detection on UNIX systems.

4.1 Intrusions

This section covers the particular security violations that are detected by NIDX.

4.1.1 Attempted External Break-in

An intruder may try to gain access to a computer system by attempting various login/password pairs until successful. Although the main purpose of NIDX is to detect intruders once they have successfully gained access, monitoring for attempted break-in is relatively easy and provides advanced warning of possible attacks on the system.

4.1.2 Attempted Internal Break-in

Many timesharing systems provide one or more methods for users, once logged in, to obtain the permissions or identification of another user. On the UNIX system, for example, gaining root permissions is the usual goal, however, intruders may try to obtain the permissions of other users. Monitoring for this type of intrusion provides advanced warnings about not only an impending attack but also *who* is responsible.

4.1.3 Browsing

Browsing is the unauthorized reading of data and is probably the most frequent offense for both internal and external intruders, but the hardest to distinguish from normal activity since most users of a UNIX system explore it regularly. NIDX monitors users for browsing in unusual, confidential, or restricted areas of the system.

4.1.4 Modification and Destruction

Malicious intruders may not be satisfied with merely browsing, they may change or destroy the objects on the system. Although direct modifications and deletions of objects are relatively easy to detect, other intrusions may be more difficult. For example, an intruder may insert a virus into a commonly used program. When that program is run by an authorized user, the virus performs some covert activity (such as changing file access permissions) and also infects more programs. Thus, a virus that is seeded by an intruder is spread unknowingly by authorized users.

4.1.5 Theft

Information theft over communication lines to external systems is a common goal of intruders who successfully break into a computer system. Further, this particular intrusion may be initiated by internal users. Popular objects of theft include tools, games, the operating system source, and proprietary software. Theft can be detected by monitoring for excessive I/O concurrently in system directories and terminal lines, automatic calling devices, or removable medium devices.

4.2 Suspicious Events

In this section we describe heuristics for accelerating the intrusion detection process. Some activities, taken as a discrete activity, are not necessarily security violations. For example, user *x* who has never logged in at night before logs in late one night. This activity is not a security violation, however; it is simply a suspicious event because user *x* has no history of logging in at night. A suspicious event is used to lower thresholds for reporting possible violations. That is, the occurrence of a suspicious event means that future activities are analyzed in smaller sample space, thus fewer activities are needed to trigger an alarm. The events which trigger this are:

4.2.1 Abnormal Login Events

If a user logs into the system at an unusual time period, terminal or network port.

4.2.2 Successful Internal Masquerade

Section 4.1.2 describes the Attempted Internal Break-in intrusion. However, a user who assumes the identity of another user should not be considered an intruder solely for that action. Indeed, the UNIX system *set-user-id* feature allows a user to enable other users to assume their identity when running certain programs. However, in that case there should always be a one-to-one mapping between identity assumption and objects accessed. Any discrepancies may be signs of an intrusion.

4.2.3 Change in Work Area

The system monitors the directories each user works in, under the assumption that users typically use the same set of directories each day. A change in work area, i.e. a change in the directory where significant work is done, may indicate only a change in assignment or, perhaps, an impending security violation. The choice of the new work area allows the rule base to determine whether or not a suspicious event has occurred.

4.2.4 Awkward Use of the Computer

Awkwardness in using a computer system is characterized by attempts to run commands that do not exist,

attempts to change directories to those that do not exist, attempts to read files that do not exist, searches of common system directories, and extensive use of the **man(1)** and/or system help commands. Even experienced computer users show some awkwardness when introduced to an unfamiliar computing environment. This is amplified by the many flavors of the UNIX system available.

A profile to measure the awkwardness of a user is kept. Assuming that the awkwardness decreases over time, then if a series of mistakes occurred that exceeded the some threshold, NIDX may infer that a successful break-in has occurred.

4.3 Knowledge Base

The knowledge base contains system specific knowledge about the target system being monitored and heuristics for detecting intrusions on the target system.

4.3.1 System Specific Knowledge

Knowledge about the system being monitored is essential for quick, accurate intrusion detection. For example, consider the */tmp* and */bin* directories: both directories can be read by all, however, whereas */tmp* can be written by all, */bin* can only be written by the system accounts *root* and *bin*. Thus, for an intrusion detection system, it is more far more interesting to notice a user writing into */bin* than into */tmp*. Without this knowledge, both directories must be treated equally.

The */etc/password* file is an example of system specific knowledge because it lists users and their home directories. Other examples are lists of files, devices, and directories describing their relationship to users, classified as follows:

read-public:	permit read by all
write-public:	permit write by all
exec-public:	permit run/search by all
read-restricted:	deny read by users
write-restricted:	deny write by users
exec-restricted:	deny run/search by users
read-ulist:	permit read by listed users
write-ulist:	permit write by listed users
exec-ulist:	permit run/search by listed users
read-ulist-res:	deny read by listed users
write-ulist-res:	deny write by listed users
exec-ulist-res:	deny run/search by listed users

Note that the first six classes of the above list can be defined independently of the particular UNIX system being monitored. Thus, they are applicable to all monitored UNIX systems. The last six classes can be populated in the knowledge base to describe access rights for a particular UNIX system if finer-grained detection is desired. See section 6.1 for the process of handling objects not listed in the knowledge base.

Also, the knowledge base includes a list of devices and types, for example the dial-up lines, the removable storage medium devices, etc. The system specific knowledge is entered and maintained by the security officer or system administrator. It can be as detailed as desired for each system being monitored.

4.3.2 Intrusion Detection Heuristics

These heuristics are rules of thumb for detecting intrusions on UNIX systems. In many cases, NIDX cannot make a simple binary decision regarding the potential threat of a particular activity and all uncertain activities cannot be flagged as violations because of the number of alarms generated. Thus, heuristics are used to determine if particular activities are typical of malicious behavior or not. Some example heuristics are:

1. Users should not read files in other users' personal directories.
2. Users must not write other users' files.
3. Users who log in after hours often access the same files they used earlier.
4. Users do not generally open disk devices directly.
5. Users should not be generally logged-in more than once to the same system.
6. Users do not make copies of system programs.
7. Writable directories are good places to put Trojan Horses.

The heuristics in NIDX have been gleaned from a variety of sources including security literature [3-6], discussions with security personnel, and in the future, from new attack techniques learned from experience.

4.4 UNIX System Audit Trail

NIDX requires a detailed trace of user activity and the objects they access. The standard UNIX system V accounting package logs every program executed, recording such information as the program name, amount of disk I/O, and CPU time. However, it does not provide a record of the objects read or written.

The UNIX system kernel must be modified to provide a trace of system calls invoked by user processes. This is necessary because only at the system call level can user interaction with system objects be traced in real-time. Not all system calls need be audited; only those that manage objects and affect user privileges are traced.

System calls traced[1] are:

access	dup	link	setgid	stat
chdir	exec	open	setuid	unlink
creat	fork	pipe	setpgrp	

A typical audit record contains, using the **open(2)** system call as an example, real and effective user/group id, process id, object, action, time stamp and success indicator. Thus, invocation of the system call:

```
open ("/usr/dsb/x", O_RDONLY);
```

results in the following trace record:

user:	195	type:	file
group:	130	flag:	read
euser:	195	status:	0
egroup:	130	errno:	0
call:	open	time:	123456789
object:	/usr/dsb/x		
object-owner:	195		
object-group:	130		

This trace is typical of what is provided by secure operating system implementations [7] however the intended audience is the intrusion detection system and not a human security officer.

4.5 NIDX Architecture

Figure 1 shows the NIDX architecture and data flow. Referring to the diagram: users interact with the target system, running programs, editing files, etc. An audit trail of system calls is transmitted, in real-time, to NIDX which is running on a computer independent of the target computer.

The system specific knowledge base, rules encoding heuristics and control strategy, and user profiles that make up the expert system are implemented in an expert system shell running on the monitoring computer. Once on the workstation, the traces of the system calls are entered into the expert system's working memory where the rules reason about them. When intruders are detected, an alarm showing the violation and an explanation of the reasoning that led to its discovery is displayed on the security administrators workstation. Then, the security administrator can instruct the target system to contain the intruder in some way. Some initial containment mechanisms under consideration include restricting the intruder to their home directory or logging them off the target system and disabling their account. Another possible action, as described in [8], is to limit the destructive damage the intruder can do then continue to monitor with the goal of learning new attack techniques that can subsequently be included in the rule base.

At the security officers' discretion, NIDX can be permitted to automatically provide containment instructions to the target system when intrusions with a very high confidence level are detected.

5. PROFILES

The profiles are used to summarize and characterize the behavior of each user on the system. Profiles are used to augment the intrusion detection process when the history of past activity would facilitate the analysis of a suspected violation. There is a set of permanent profiles for each user that are updated at the end of each login session to reflect that day's use of the computer. Separate session profiles are kept while the user is logged-in to distinguish current from past activities. This section describes the initial set of NIDX user profiles. Because profiles are computationally expensive to use and have significant storage requirements, part of the on-going research is to determine which profiles are necessary for effective intrusion detection.

5.1 Successful Login Profiles

5.1.1 External Login

Login profiles are organized primarily by time of day because, generally, most users have a consistent work schedule. There are login profiles for various time periods including week-day, evening, late-night, weekend-day, and weekend night. Each profile lists the access method, e.g. hard-wired line or dialup. This information provides a consistent record of each users' general computer access habits.

5.1.2 Internal Login

The only way for a user to "login-in" as another user once on the system is via the **su(1)** command, given that the user does not already have super-user permissions. Successful and failed attempts are logged currently in */usr/adm/sulog*.

A profile is maintained for each user and the logins that user successfully **su**'s to. The profile includes user id, new user id and the work area directory profiles used while under the new id. This profile is necessary to distinguish assumption of user id via the **su** command versus a set-user id program.

1. Readers familiar with UNIX System V release 3.1 will notice that the message queue, semaphore, shared memory, and Transport Interface (TI) system calls have been omitted. Auditing them is necessary for complete intrusion detection, but, because the techniques needed to monitor them are basically the same as the listed system calls, they are not described.

5.2 Attempted Login Profiles

This profile records the number of attempted logins for users.

5.2.1 External Login

Intruders may use trial and error in choosing user ids and passwords when attempting to break into a system. Some useful user ids that may be tried are *guest* or *sys*. Tracking the user ids entered, device and time of day may be useful in detecting patterns of attempted break-ins. The profile includes user, initial time stamp, average number of attempts per day, number of attempts in the last *n* minutes and devices used.

5.2.2 Internal Login

A record is kept for each user and their failed attempts to assume another id. Note that only failed **su** attempts aimed at real users are recorded since the password file lists valid user names and is readable by all. An internal-login-failure profile includes user id, and attempted user name.

5.3 File and Directory Access Profiles

Every object in the UNIX system is assigned an owner when it is created and, if desired, the ownership can be changed. The file and directory access profiles maintained for each user describe that user's relationship to the files and directories owned by others. For example, the profile *dsb-read-kob* is updated whenever user dsb reads a file or directory owned by the user kob. Obviously, there are too many objects on a system to monitor each individually. Thus, objects are grouped into classes and each class is monitored rather than individual objects.

The following sections describe the actual profiles. The '*' below denotes the actions performed on the objects. The actions monitored are: read, write, execute, search, and delete. The read profile is updated whenever a file is opened for reading. The write profile is updated whenever a file or directory is opened for writing. The execute profile is updated whenever a program is invoked. The search profile is updated whenever a directory is opened for reading. The delete profile is updated whenever a directory or file is removed. Each profile includes the following information user id, profile type, e.g. *user-read-other*, the number of successful actions, the number of unsuccessful actions, cumulative and session means.

5.3.1 User--other*

These profiles monitor one users interaction with another user's objects. Note that "another user" does not include system objects, which are monitored separately. Profiles can be created which monitor a users interaction with one other user's objects, e.g. *user-*-xyz*, or which monitor a users interaction with several user's objects, e.g. *user-*-abc,hij*.

5.3.2 User--self*

These profiles monitor a users actions to objects he/she owns. Although these profiles may not appear to be useful, they may be used to detect masquerade violations.

5.3.3 User--system*

These profiles monitor the use of uncommonly used or restricted system objects that, if accessed by a user, may result in a security violation. Examples are devices in **/dev**, *uucp* files in **/usr/lib/uucp**, and the system source.

5.4 Work Area Profiles

It is useful to know over a given time where a person works, in terms of directories on the system. A user's work areas are the directories he/she accesses during a login session. One way to focus the attention of the intrusion detection system is to notice that user is not working in a previously established work area.

A work area profile is an ordered list of all the directories a person accesses over a given time with those directories most recently accessed at the front of the list. Although user's primary work areas may change over time as his/her tasks and skills change, deviations allow the intrusion detector to focus on activities occurring in the new directories. Thus, this profile is not used as the sole basis for detecting an intruder; rather it is used to detect suspicious events.

5.5 User Id and Group Id Profiles

All *set-user/group-id* programs on a particular system should be known to the system administrator because each is a a potential source or target for security violations. Except for a few system commands, such as **su(1)**, **mkdir(1)**, and **rmdir(1)**, *set-user/group-id* programs are rare and should only be used to provide access to files not available without special permission. Thus, each non-system *set-user/group-id* program will be profiled independently for all users. That is, this profile set covers the *set-user/group-id* programs that each individual user invokes. Deviations in the directories accessed or programs run from a *set-user/group-id* program may be indications of an intrusion.

5.6 Awkwardness Profile

Assuming that regular users make less *mistakes* and require less help than irregular users, this profile may be useful in detecting an intruder who is unfamiliar with the system he/she has broken into. A *mistake* is defined as a program invoked that is not found or a directory or file accessed that does not exist. Extensive

searches of common system directories and extensive use of the system **help(1)** or **man(1)** commands are also activities indicative of irregular users and, thus, are monitored by the awkwardness profile.

The profile includes user, average number of awkward events per session hour, number of awkward events this session.

6. INTRUSION DETECTION

This section describes two issues resulting from knowledge-based intrusion detection and presents the techniques developed to resolve them. The first shows how the knowledge base and heuristics are used to distinguish authorized from unauthorized behavior. The second explains how inference of particular activities from the detailed audit trail is done.

6.1 Detecting Unauthorized Browse

Detecting unauthorized browse is done in two parts: first, detecting browse activity from the audit trail and, second, determining authorization. Consider the following user command:

```
$ cat /usr/kob/x
```

On a UNIX system the directory /usr/kob is typically the home directory of the user kob. Let us assume this command was entered by some other user, say user dsb. The system call trace (abbreviated) is:

```
call: open object: /usr/kob/x    flag: read
```

Detection of browse is relatively easy since browse activity results from opening files and directories for reading. Section 6.2 shows how browse is distinguished from other activity also involving reading, e.g., copying or theft. Each read audit record that does not correspond to copy or other activity is individually authorized according to the following procedure:

1. If the object is */etc/password* or another object that is listed as *read-public* in the knowledge base then, read is authorized.
2. If the object is listed as *read-restricted* and if user 195 is not a system user id, then read is not authorized and an alarm is generated.
3. If the user and object are explicitly listed in the *read-ulist* class then read is authorized. If the user and object are explicitly listed in the *read-ulist-res* class then an alarm is generated.
4. If there is no explicit read authorization knowledge for this user and object, then heuristic authorization is done. The remaining list items give examples of heuristic authorization. First, if the object is owned by the user and no suspicious events have occurred, then read is authorized.
5. If the object is owned by the user and one or more suspicious events have occurred and the object is not a directory listed in the user's recent work-area-profile, then an anomaly record is generated. Several anomaly records are needed before an alarm is generated.
6. If the object is in the directory hierarchy of another user, say user *x*, and is not in a "generally public readable" directory, e.g., *~x/bin*[2] or *~x/rje*, then an anomaly record is generated. Specification of "generally public readable" directories is contained in the knowledge base.
7. The previous heuristic can be augmented with the use of profiles. For example, authorization may be granted if user 195 or members of user 195's group had shown previous history of reading that particular object.

This example has shown how some simple intrusions can be detected from just one audit trail record. The list contains a few of the many heuristics used to determine authorization of browse or read activity when simple binary decisions cannot be made. Part of the ongoing research is to generalize and combine heuristics.

6.2 Detecting Copy Intrusions

Distinguishing copy intrusions from browse intrusions is important because once a file is copied, the copier has unlimited access to it. Also, since devices are actually special files on UNIX systems, the command:

```
$ cp /usr/kob/x /dev/diskette1
```

copies the file to floppy disk which can result in information theft. Copy intrusions are detected the same way as browse intrusions: first the copy is detected, then authorization is determined. Authorization is done using knowledge of user/object relationships and heuristics in the same manner as browse intrusions, however, on UNIX systems, detection that copy activity has occurred is difficult for two reasons. First, several audit records are needed to determine that potential copy activity has occurred, and second, there are many methods to copy a file each resulting in a potentially different audit trail.

Consider the following three examples. Each shows a command that copies the file /usr/kob/x into the file /usr/dsb/y and its respective audit trail (abbreviated).

2. The symbol *~x* is an abbreviation for the home directory of user *x*.

Example 1

```
S cp /usr/kob/x /usr/dsb/y

call: exec  object: /bin/cat
call: open object: /usr/dsb/y    flag: write
call: open object: /usr/kob/x    flag: read
```

Example 2

```
S cat < /usr/kob/x > /usr/dsb/y

call: open object: /usr/kob/x    flag: read
call: open object: /usr/dsb/y    flag: write
call: fork
call: dup  object: stdin
call: dup  object: stdout
call: exec object: /bin/cat
```

Example 3

```
S cat /usr/kob/x | cat > /usr/dsb/y

call: pipe
call: open object: /usr/dsb/y    flag: write
call: fork
call: dup  object: stdin
call: dup  object: stdout
call: exec object: /bin/cat
call: fork
call: dup  object: stdout
call: exec object: /bin/cat
call: open object: /usr/kob/x    flag: read
```

Although, the audit trail for each command is different, some re-occurring themes can be exploited. Each eventually opens */usr/kob/x* for reading and */usr/dsb/y* for writing. Also, the audit trail patterns *fork-dup-exec* and *pipe-fork-dup-exec* are indications of I/O redirection.

This knowledge is used with the following technique to detect copy: Audit records are processed using forward-chaining strategy with the goal of discovering either a "restricted"[3] file opened for reading or an "unrestricted" file opened for writing. If that goal is reached, the goal state *detected-copy* is set and a subgoal is created to substantiate it. Satisfying the subgoal depends on whether opening a restricted or unrestricted file satisfied the *detected-copy* goal state. A subsequent unrestricted file must be opened for writing to satisfy the subgoal created when a restricted file was opened for reading. Conversely, if the *detected-copy* goal state was reached when an unrestricted file was opened for writing, then a restricted file must be later be opened for reading to satisfy the subgoal. Satisfying the subgoal is done using a backward-chaining strategy.

If the process exits before the subgoal is reached or another violation is detected, then the *detected-copy* goal and the subgoal are removed from consideration. If patterns for I/O redirection are detected in the audit trail, meaning that the source of all I/O has not yet been determined, then analysis must continue.

3. With respect to the user generating the audit trail.

7. CONCLUSION

Contemporary security tools provide various levels of protection against potential intruders. However, this protection is given only as long as these preventive mechanisms are not compromised. Once defeated by a sufficiently determined intruder, the underlying system is again left open to tampering. Moreover, this tampering is often untraceable by conventional means. Thompson, for example, demonstrates in [9] how clever system programmers can insert devices into system software that are virtually undetectable by current security mechanisms. A complementary mechanism to prevention is detection. Conventionally, this is achieved by providing detailed audit data that allows security officers to detect intrusions, assess the damage and repair the security holes. The approach described in this paper is an extension of this batch mode audit trail inspection, which is designed to analyze the audit trail in real-time using expert system technology. A key objective of this approach is to allow real-time containment of the intruder's activity.

We have described the architecture and design of the NIDX expert system for network intrusion detection. The NIDX system, specifically targeted for the UNIX System V environment, detects security violations in real-time by analyzing a trace of UNIX system calls using system-specific knowledge, history profiles of users' activities, and heuristics about typical intrusions and attack techniques.

Finally, there are several open issues we are investigating: It is unreasonable to expect a general purpose computing system to provide sufficient resources for its users while monitoring itself for intruders. Even the generation and transmission of the audit trail can use a significant amount of CPU time. Consequently, we are investigating the role of parallel processing in intrusion detection where additional processors are used to offload the auditing and monitoring functions from the target system. In addition, we are investigating the tradeoffs between fast detection and the number of false

alarms generated. An acceptable balance can be found using intrusion detection heuristics augmented with system specific knowledge and user profiles tuned from extensive field studies. Finally, although the initial version of NIDX is targeted for the UNIX System V environment, we are planning to apply this approach other operating systems

REFERENCES

[1] D. E. Denning and P. G. Neumann, *Requirements and Model for IDES - A Real-time Intrusion-Detection Expert System,* SRI Project 6169, SRI International, Menlo Park, CA, August 1985.

[2] D. E. Denning, D. E. Edwards, R. Jagannathan, T. F. Lunt, and P. D. Neumann, *A Prototype IDES: A Real-Time Intrusion-Detection Expert System*, Final Report, SRI Project ECU 7508, SRI International, Menlo Park, CA, August 1987.

[3] J. Lobel, *Foiling the System Breakers,* McGraw-Hill Book Company, 1986.

[4] R. Farrow, *Security For Superusers,* UNIX/WORLD, May 1986, pp 65.

[5] P. H. Wood and S. G. Kochan, *UNIX System Security,* Hayden Books, 1985.

[6] R. Farrow, *What Price System Security?,* UNIX/WORLD, June 1987, pp 54.

[7] V. D. Gligor, E. L. Burch, C. S. Chandersekaran, R. S. Chapman, L. J. Dotterer, M. S. Hecht, W. D. Jiang, G. L. Luckenbaugh, and N. Vasudevan, *On the Design and the Implementation of Secure Xenix Workstations,* Proceedings of the 1986 IEEE Symposium on Security and Privacy, April 1987, pp 102.

[8] C. Stoll, *What Do You Feed A Trojan Horse?,* Proceedings of the 10th National Computer Security Conference, September 1987, pp 231.

[9] K. Thompson, *Reflections on Trusting Trust,* ACM Turing Award Lecture, Communications of the ACM, August 1984.

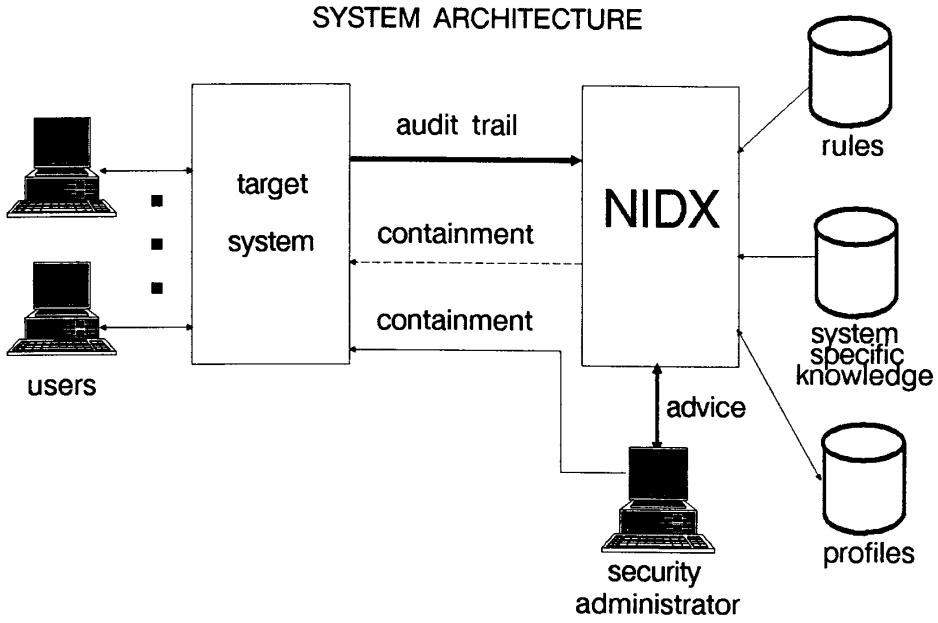

Figure 1