Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring *†

Calvin Ko George Fink Karl Levitt
Department of Computer Science
University of California, Davis
Davis, CA 95616
{ko, gfink, levitt}@cs.ucdavis.edu

Abstract

We present a method for detecting exploitations of vulnerabilities in privileged programs by monitoring their execution using audit trials, where the monitoring is with respect to specifications of the security-relevant behavior of the programs. Our work is motivated by the intrusion detection paradigm, but is an attempt to avoid ad hoc approaches to codifying misuse behavior. Our approach is based on the observation that although privileged programs can be exploited (due to errors) to cause security compromise in systems because of the privileges accorded to them, the intended behavior of privileged programs is, of course, limited and benign. The key, then is to specify the intended behavior (i.e., the program policy) and to detect any action by privileged program that is outside the intended behavior and that imperils security. We describe a program policy specification language, which is based on simple predicate logic and regular expressions. In addition, we present specifications of privileged programs in Unix, and a prototype execution monitor for analyzing audit trails with respect to these specifications. The program policies are surprisingly concise and clear, and in addition, capable of detecting exploitations of known vulnerabilities in these programs. Although our work has been motivated by the known vulnerabilities in Unix, we believe that by tightly restricting the behavior of all privileged programs, exploitations of unknown vulnerabilities can be detected. As a check on the specifications, work is in progress on verifying them with respect to an abstract security policy.

*This work is funded in part by the National Security Agency University Research Program under Contract No. DOD-MDA904-93-C4083 and by ARPA under Contract No. USNN00014-94-1-0065.

†Published in the 1994 Computer Security Application Conference

1 Introduction

Computer systems are vulnerable to attacks. Despite the best effort to uncover and remove security errors, vulnerabilities in computer systems still exist, enabling outside attackers to gain entry to systems and inside attackers to exploit their privileges [3, 4].

Vulnerabilities in privileged programs (e.g., setuid root programs in Unix, such as rdist, sendmail, and fingerd) have been one of the major techniques for attackers to obtain necessary privileges to accomplish their missions. These programs run with high privileges that allow them to bypass the kernel protection mechanism, in effect violating the system policy. In principle, they are designed and trusted not to imperil the security of the system, but due to errors, they can be used to bypass security safeguards [2, 21]. For example, during its testing, a backdoor was inadvertently inserted into the BSD sendmail program, enabling users to obtain root privileges. As another example, the finger daemon program neglects to limit the size of a input string, enabling an attacker to overflow its buffer to obtain root access in the host providing the finger service. [7, 19]. Often, such errors are subtle, and the exploitation involves multiple processes interacting in unexpected ways. Therefore, these errors are often not detected during testing and not discovered until long after system releases.

In this paper, we discuss a technique for detecting exploitations of vulnerabilities in privileged programs. Our approach employs specifications of the security-relevant behavior of privileged programs as an oracle for comparison with actual program behavior recorded in system audit trails. Our approach is based on the observations that although privileged programs have the potential to do "anything" because they often run as setuid root, the intended behavior is, of course, limited and benign. The security-relevant part of the intended behavior of these programs are rather simple and can be specified in a precise and concise manner, typically in a few lines of predicate logic. Most exploitations of vulnerabilities in these programs involve "tricking" these programs into violating their intended behavior. We describe a specification language for the privileged programs (setuid root programs and daemons) in Unix. Using just the audit trails generated by the operating system, we can detect actions of most privileged programs which are in violations of the specified behavior. In addition, the concise specifications of the privileged programs and the low-complexity audit analysis algorithm enable real-time monitoring of these programs. With the specifications, we are able to detect known attacks which exploit vulnerabilities in these programs. Although the specifications are written with the knowledge of the vulnerabilities, we strongly believe that unknown vulnerabilities in privileged programs can also be detected by our method.

Our approach is a variant of intrusion detection [6], wherein audit trails are analyzed in real time to detect ongoing attacks. Currently, intrusion detection employs statistical modeling of normal (user) behavior [6, 22, 17] and rule-based modeling of suspicious user behavior [10, 18, 16]. For the purpose of intrusion detection, user behavior is considered suspicious if it bears similarity to known attacks or known attack methods, or is in direct violation of the system policy. While one can in principle codify the steps to exploit known vulnerabilities for purpose of inclusion in an intrusion detection system, it is much more difficult to cope with unknown vulnerabilities. Our approach is essentially a specification-based model of intended behavior of privileged programs, which is a more systematic way to identify "misuses" of privileged programs, and is capable of detecting attacks which exploit unknown vulnerabilities in these programs. Therefore, we anticipate immediate application of our approach to intrusion detection systems.

The paper is organized as follow. To motivate our work, Section 2 presents a few examples of the known vulnerabilities in privileged programs in Unix. Section 3 presents our approach to modeling the behavior of privileged programs. Section 4 discusses the security specification of privileged programs. Section 5 presents our approach to monitoring the execution of privileged program using audit trails. Section 6 presents some example program policy specifications. Section 7 is related work, and Section 8 is our conclusions and recommendations for future work.

2 Examples

This section presents informal descriptions of a few attacks that exploit vulnerabilities in Unix privileged programs.

The first program we describe is *rdist* [1] (Remote File Distribute Program) [20], which is used for maintaining consistency of files on multiple hosts in a network. The single executable actually contains the client and the server. *rdist* is normally invoked by a user in a master host to distribute copies of files to remote hosts. To perform an update, *rdist* (client) invokes *rdist* (server) in the remote host using the remote shell protocol. The client and the server communicate via a simple internal protocol. *rdist* is a setuid root program because it uses privileged ports for authentication. To distinguish the client and server, we use *rdist[s]* to denote the server.

rdist contains a vulnerability which lets a user (unprivileged) in the system change the permission mode of *any* file [1]. It has been used by attackers to set the setuid bit of a system shell (e.g., */bin/sh*). The vulnerability relates to the way that *rdist[s]* updates a file. To update a file, *rdist[s]* first creates a temporary file (Fig. 1a); then, it copies data to the temporary file (Fig. 1b); when all the data is copied, *rdist[s]* changes the ownership and permission mode of the file to correspond to the master file on the remote system (Fig. 1c) ; finally, it renames the temporary file to the destination file (Fig. 1d).

The way to exploit the vulnerability is as follows: A normal user invokes *rdist* with appropriate commands to update one of his own files in the local host, in effect causing *rdist[s]* to update the file. Right after *rdist[s]* creates the temporary file and before it finishes copying data, the user renames the temporary file and creates a symbolic link (which has the same name as the temporary) but to the target file the permission of which he desires to change (Fig. 1e) [2]. When *rdist[s]* finishes copying the data, it changes the ownership and the permission of the temporary file using `chown` and `chmod`. Both of them take a symbolic pathname as parameters. The tricky point is that `chown` does not follow symbolic links, but `chmod` does. Therefore, the effect is the ownership of the symbolic link itself is changed but the insidious part is that the permission

[1] *rdist* was first released as part of 4.3 BSD UNIX, and has been very widely used by many system administrators to automate software distribution in a network environment.

[2] An attacker can have a better control of the timing and the final permission mode of the target file if he invokes the *rdist* server directly and gives internal commands to the *rdist* server to update a file.

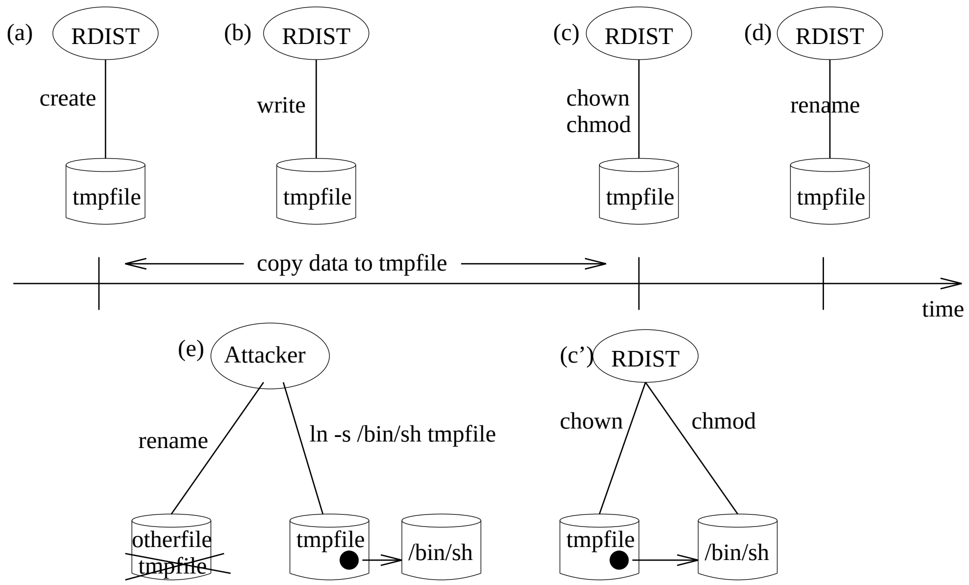

Figure 1: Visualization of a rdist Attack

mode of *"/bin/sh"* is changed (Fig. 1c').

The vulnerability we described is very subtle, involving concurrent interactions of processes, the semantics of pathname, and temporary files. One might expect, then, that the bug might not be discovered through conventional testing, indeed *rdist* was released and used for years before the error was discovered. However, in this attack, *rdist* is tricked into doing something outside the behavior intended for it. Specifically, *rdist* is used by a user to update his files in a host; therefore, when invoked by a user, the process associated with the invocation should update only the file owned by the user. In addition, *rdist* only changes the permissions of the temporary files it creates, not other files. As we discuss in Section 7, if *rdist* causes changes to the security state other than as we described, it is being exploited. Audit trail analysis can uncover such exploitative behavior.

Our second example relates to a flaw in the finger daemon, which has been exploited to obtain a root shell in the host running the daemon. Specifically, the finger daemon reads finger requests using the *gets* library call, which does not specify a maximum buffer length [3]. To exploit the vulnerability, an attacker sends a long request message to the finger daemon that overwrites the read buffer, which is in the run-time stack structure of the process. He overwrites the stack with his own code, and the return address in *gets*'s stack frame to point the code he injectes. When the subroutine returns, it branches into that buffer and executes the attacker's code. But, the intended security-relevant behavior of finger daemon is very restrictive. It should execute only the finger program (/usr/ucb/finger), and read only some status files (e.g., /etc/utmp, .plan, .profile).

[3]In current C language implementations, the return address of a function call is saved in the run-time stack.

Our third example is *Sendmail*. This single program actually serves different roles. First, it is invoked directly as a delivery intermediary by the front end mailer (/usr/ucb/mail). It runs as a daemon process to accept mail arriving at the mail port, and routes mail to remote systems. In delivery of mail, it also executes mail handlers (e.g., "vacation" programs) for users. In addition, it retries pending mail in the mail queue periodically. When invoked as *mailq*, it displays the contents of the mail queue. Also, it can be invoked to rebuild the alias database.

There have been quite a few vulnerabilities associated with different distributions of *Sendmail*. One of earlier vulnerabilities was the presence of a backdoor in the Berkeley version, which was included for testing purpose. The backdoor enables any user connected to the mail port to obtain a shell running as root by entering a special command. Recently, another vulnerability was discovered which enables a normal user to cause root to execute a program he specifies, in effect acquiring root access. Despite the complexity of *Sendmail*, its intended security-relevant behavior is well defined and relatively simple to specify. Sendmail should manipulate only files inside the mail spool directory and inside the mail queue directory. In addi-

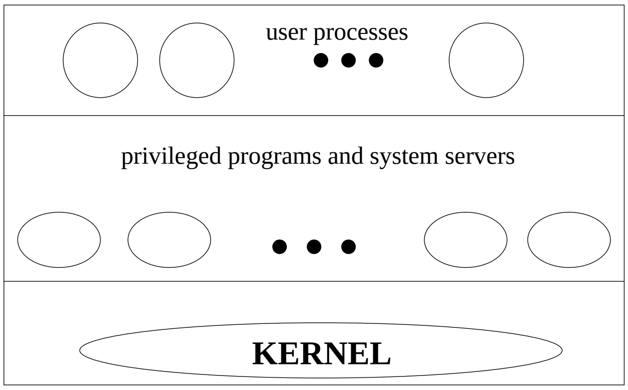

Figure 2: The System Model

tion, it may write to some configuration files in the /etc directory, where the file names are prefixed by "sendmail". No other file operations are intended.

3 Modeling the Behavior of Privileged Programs

Figure 2 depicts a simple model of Unix, which is also applicable to numerous other operating systems. The bottom level is the operating system kernel, which manages all resources (e.g., memory, disks, files, cpu). Resources can only be accessed through invocations of system calls. A kernel normally provides some mechanisms for mandatory and discretionary protection of the resource it manages. Thus, actions of user processes (top level) are restricted by the kernel to prevent any security violation. However, a system normally consists of "privileged" processes (middle level), which are allowed to bypass the kernel's security mechanism in order to accomplish their jobs. They could be part of the kernel, but reside outside to keep the already complex kernel from being even bigger, and also to allow these programs to be portable among different proprietory kernels. Nevertheless, they are trusted not to imperil the security of the system. The term "privileged program" refers to the program a privileged process executes. In current systems, these programs exist in two forms: amplification program and server program. The term "amplification" indicates that the privileges associated with the process is amplified when it executes an amplification program (e.g., setuid program in Unix). A server program runs in background, listens to request from users, and serves the requests on behalf of the user. They normally are invoked at system startup or invoked by a super server process on requests (e.g., inetd). Such

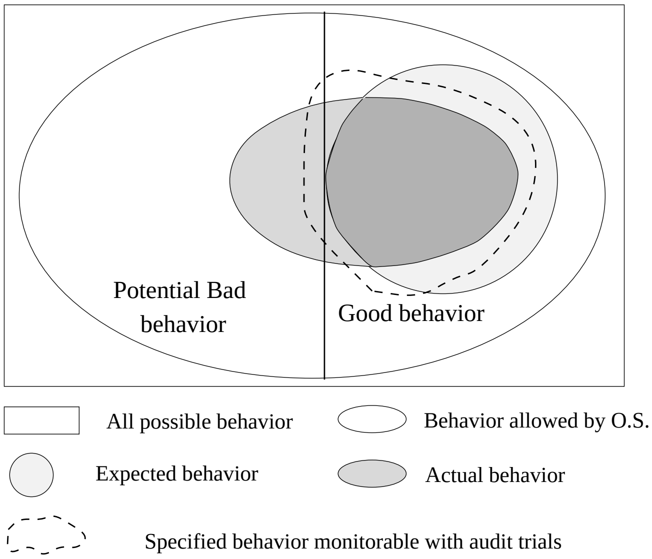

Figure 3: A Behavior Model of Privileged Programs

programs are almost always given more power than they need to accomplish their jobs because of the limitations of the access control mechanisms of the operating system; that is most operating systems provide protection at too coarse a granularity to preclude the need for privileged processes or programs.

Figure 3 illustrates the problems with these privileged programs and our approach to cope with such problems. The rectangle represents all possible behavior of a privileged program. The largest ellipse represents the behavior allowed by the kernel. This allowable behavior is divided into two portions: "good" behavior and "bad" behavior, which illustrates the fact that a privileged program is given the authority to do bad things. In principle, the behavior of a privileged program should not compromise security; that is, its intended behavior (the shaded circle) lies completely inside the good behavior portion of the ellipse. However, the actual behavior of the privileged program (the shaded ellipse) may deviate from the good behavior because of program errors and incorrect setup. Our solution to the (perhaps) inevitable presence of such errors is to specify the security-relevant behavior of each privileged program, which is monitorable using audit trails (dotted lines).

One important aspect of the behavior of a privileged program is its set of allowed accesses. More formally, an access is an order pair (op_type, obj), where "op_type" is the access right or the operation type (e.g., read, write, execute, chmod) and "obj" is the object (e.g., a file, a directory). Although it cannot guarantee perfect security, we find that by specifying the set of accesses allowed, the behavior of these privileged programs can be restricted in a way that the probability of security compromise is greatly reduced. As the specification is concerned with the access rights a program holds, we call the set of allowed accesses a *program policy* or just *policy*.

3.1 The Scope of Privileged Program Execution

An execution of a privileged program may not be confined to only one process. Often, a process running a privileged program may create another process to perform part of the job. This situation is very common in Unix daemon programs, wherein a daemon forks a child process to handle each request, and then continues to listen for other incoming requests. From our prospective, the child process is considered to be a part of the execution of the privileged program associated with its parent.

In addition, a privileged program may invoke other non-privileged programs to accomplish part of the job. Although the control is passed to the invoked program, the execution is still considered to be part of the privileged program execution. One tricky aspect occurs when a privileged program invokes another privileged program. For instance, *Sendmail* invokes */bin/mail*, the backend mailer, to deliver mail to users. There are two ways to view this situation, equivalent with respect to assurance of security: (1) the execution of the new privileged program is consider to be part of the execution of the original privileged program, (2) the execution of the original privileged program ends, followed by the execution of the new privileged program. In our model, we consider the execution of the original privileged program continues to include the invoked program, regardless the invoked program is privileged or not.

The execution of a privileged program within a process terminates either (1) the process dies, or (2) the process gives up its privileges permanently. For instance, many privileged programs in Unix provide a shell escape feature which enables a user to obtain a shell. Right before it executes the shell and passes the control to the user, it changes the uid and gid of the process to that of the user.

3.2 Relation to Users

In existing computer systems, privileged programs can be invoked directly or indirectly by a user. For instance, in Unix, a user invokes *passwd* (the password program) directly to change his password stored in the password file. On the other hand, consider the case when a user invokes *lpr* to request that a file be printed; in turn *lpr* communicates with *lpq*, which prints the file. In this case the actions of the *lpq* are invoked indirectly by the user.

The set of allowed accesses for an execution instance of a privileged program is a function of the user who invoked the program. More precisely, we consider a subject to be the pair (user, privileged_program). The idea is similar to the integrity policy of Clark and Wilson [5] in which high integrity data can only be accessed by authorized users using a particular program (Transformation Procedure). The concept of a user or a process as a subject in an operating system is well understood. Here, we employ the pair (user, program) as a subject. The entities which actually perform operations are processes. A process is associated with a user (denoted by a uid in Unix), who is accountable for the actions of the process. An operating system usually restricts the accesses of a process based on its associated user. But, a process at any time is executing a program. Therefore, (U, P) is the process which is owned by the user U and executing the program P.

4 Specification of Program Policies

In this section, we present our approach to the specification of privileged program policies. We first identify some properties of privileged programs (i.e., their policies), and then we present a language for specifying the policies. Briefly, we discuss a method for reasoning about programs.

4.1 Properties of Privileged Program behavior

We present some properties of interest for the privileged programs to motivate the features we include in our specification language. The set of objects a privileged program can access depends on the associated user. For instance, a user invokes **rdist** to update his files in remote hosts. In general, objects accessible to a program are a function of the user invoking the program. Below, we identify some security-relevant properties of program behavior and present an approach to specifying them.

- Name Oriented: In an operating system, a program running in user space requests system services through system calls in which an object (e.g., a file, a printer) is identified by a symbolic name (e.g., pathname). The set of objects (e.g., files, ports) that a privileged program can access with a particular operation is often *name oriented.* For instance, some privileged programs (e.g., **lpr**, **lpq** in Unix) are permitted to access files inside a particular directory (e.g., printer spool directory), where the pathnames for the files are prefixed by the pathname of the directory. As another example, the home directory of a user is often given the same name as the user (e.g., /usr/home/ko is the home directory of user ko). We use regular expressions to model the name space of the objects. Regular expressions are well-suited to represent a set of strings such as the set of names for resources. For example, the regular expression "$/var/spool/(*/)^**$" matches every file inside the directory "$/var/spool$".

- Conditional: Whether an object is accessible often depends on the attributes of the object, e.g., the owner, the permission mode. For instance, when a user invokes **lpr** to print a file, **lpr** is able to read files readable by the user. A readable file is either owned by the user and owner-readable, in the same group as the user and group-readable, or world-readable. To express this kind of relation, we use a predicate construct similar to that in Prolog, where an object is a variable, and a formula is used to bind the value of the variable (including the value of all its attributes).

- Abstract State of the program execution and system: In some cases, what a program can do depends on the current state of the program execution. For instance, many programs create a temporary file; therefore, such a program should be allowed to read, write, and delete the temporary file it created. We represent the relevant abstract state of program execution in terms of values bound to state variables. Drawing on our examples, one variable is the set of files created by the program. In Unix, other state variables could be the effective uid and the effective group id of the process executing the program. An important consideration for our purpose is that an abstract state of program execution or of the system is meaningful only if it is extractable from audit trails. Currently, specifying and tracking of the states are done in an ad hoc manner. But, a language for defining abstract states and tracking of them from audit trails is being developed.

4.2 A Program Policy Specification Language

The goal of the policy specification language is to provide a simple way to specify the policies of privileged programs. In addition, the language should be easy to translate to rules that operate on audit trails, thus permitting an execution of a program revealed by audit records to be checked against the program's specification. Furthermore, the specification should be sufficiently formal to allow its verification with respect to an overall system policy. We describe the policy specification language informally and primarily through examples. A more detail description of the syntax and semantics of the language is in [14]

Our language is based on predicate logic and regular expressions. The alphabet of the specification language is derived from the system to be modeled. Of particular interest is the set of operations O (e.g., read, write, exec) and their parameters. For instance, in our Unix model, a read operation takes a file as its only parameter. The set of object types T (e.g., file, printer, port) and the attributes associated with each type of objects are also relevant here. (For example, the attribute of a file can be name, owner uid, permission mode.) Last, the abstract state variables are also of interest. The alphabet of the specification language consists of

- a set of operation predicate symbols $OP = \{\mathtt{op}, \sim \mathtt{op} | op \in O\}$.
- for each type $t \in T$, a set of attribute symbols A_t.
- a set of state variable symbols S.

A program policy of the program *myprog* takes the following form:

```
PROGRAM myprog(U)
     rule₁
     ...
     ruleₖ
END
```

U is a parameter referring to the user associated with the execution of the program. The body of a program policy specification is a list of rules that characterize the set of parameter values allowed for each operation $op \in OP$. A rule is similar to a predicate definition in Prolog. For instance `read("/etc/passwd")` and `read(F) :- F.ouid = U.uid` are rules which define the value of the predicate `read`. In this example, `read` is a predicate corresponding to the operation *read*, which takes a file as parameter. A file can be identified by its pathname. The first rule says that *read* is true for the file "/etc/passwd". In the second rule, the parameter to read is a variable, whose value is bound by the formula following it ((F.ouid=U.uid)). Therefore, the second rule indicates that read is true for any file, where the file owner is the user associated with the program execution. In this case, ouid, the owner uid of the file, and uid, the uid of the associated user are attribute symbols in A_{file}, and in A_{user} respectively.

The set of parameter values allowed for an operation *op* is determined by the value of the predicate op and $\sim$ op. An operation *op* is allowed for the parameters $p_1, p_2, ..., p_k$, iff $\mathrm{op}(p_1, p_2, ..., p_k)$ is true and $\sim \mathrm{op}(p_1, p_2, ..., p_k)$ is false.

Figure 4 shows an example specification for a simple system, where the operations are *read*, *write*, and *exec*, all taking a file as parameter. A file is identified by its name (path). Rule (1) indicates that the program can access the file, "*/bin/ls*", with the *exec* operation. In rule (2), the "+" operator concatenates two strings; hence, the program can write to the file "/usr/spool/mail/*username*". The "=~" symbol in rule (3) is the regular pattern matching operator. Rule (3) allows the program to execute any file whose name matches the regular expression "/bin/((sh)|(csh))". In other words, the program is allowed to execute "*/bin/sh*" and "*/bin/csh*". In rule (4), the special symbol "*" matches with any object of any type, which means that the program is allowed to read any file. However, rule (5) defines the read predicate, which specifies that the program is not allowed to read "*/etc/passwd*". Lastly, rule (6) consists of a built-in predicate inside, which returns true if F is inside "*/etc*".

4.2.1 A Specification Language for Unix

We present a specification language for a Unix system, in which three object types are considered: user, file and port. Figure 5 provides four tables which summarize the attributes of each type of object, the set of operations, some relevant state variables, and some of the built-in predicates and their meanings.

With this specification language, we created specifications for the Unix setuid programs and network daemons. In practice, some details of a program policy may be site dependent. For example, the set of users, and the directory for storing user mail may differ among hosts. To be able to have a single set of program policies applicable to all sites, one can identify the places in a generic policy that are site-specific and replace them with parameters. We provide this feature using the C macro preprocessor with site-specific parameters provided by a list of "#define" statements. To instantiate a policy, one can run the policy and the symbol definitions through a macro preprocessor.

```
set of object types = {user(name, uid),
                       file(name, ouid, pmode)}
set of operations = {read(file), write(file), exec(file)}

PROGRAM myprogram(U)
  exec("/bin/ls");                              - (1)
  write("/usr/spool/mail/" + U.name);           - (2)
  exec(F) :- F.name =~ /bin/[(sh)|(csh)]";      - (3)
  read(*);                                      - (4)
  ~read("/etc/passwd");                         - (5)
  write(F) :- inside(F, "/etc");                - (6)
END
```

Figure 4: An Example Program Policy in a simple system

4.3 Reasoning about the specifications

Our goal is to write specifications for privileged programs, having each specification reflect the "least privilege principle" [11] to restrict the access rights of these programs. There are two important questions related to our approach. First, how can we determine intended behavior of a program and write the security specification? Second, is the specification of the program correct? Currently, we specify setuid programs based on the functionality of the program and some global security goals (e.g., system integrity, data integrity, and data confidentiality). The policies of the setuid programs we have specified are based on the goal of system data integrity generally related to the following, among others: passwords, mount tables, security attributes of users, user table. However, one would come up with a different set of specifications for a goal of confidentiality, and a different set for denial of service.

We are developing a Unix security model which aims at providing a basis for writing security specifications of the privileged program and formally checking the correctness of the specifications. Specifically, the model should permit a determination of whether security can be compromised, and of course, the way they could be compromised, with the assumption that the privileged program implementations conform to our

Object type	Attribute			
user	name:string	uid:integer	gname:string	gid:integer
file	name:string	ouid:integer	ogid:integer	pmode:integer
port	num:integer	priv_port:boolean		

Type of Operations		
read(file)	write(file)	exec(file)
chmod(file, [integer])	unlink(file)	link(file, [file])
chown(file, [user])	create(file)	bind(port)

State Variable	Explanation
Filecreated	Set of files created by the program
Euid	Effective uid of the current process
Egid	Effective gid of the current process

Name	Explanation	Meaning
own(U,F)	U is the owner of F	U.uid = F.ouid
inside(F,D)	F is inside directory D	F.name = D.name + ”(∗/)∗∗”
worldwritable(F)	F is worldwritable	F.pmode & 0002 = 0
create(F)	F is created by the current program	F.name = Filecreated

Figure 5: A Specification Language for Unix

specifications. Our model consist of the specification of the kernel and an abstract state description of the system. Briefly, the kernel provides a set of operations (i.e., system calls) for users, and the privileged programs provide additional operations described by their specifications. The kernel and all the privileged programs together are intended to enforce a system policy. We attempt to define a global policy for a whole system, such as Unix, and reason about whether the specifications satisfy this policy.

5 Real-time Security Monitoring

Starting with the security specifications of the privileged programs, we present our approach to monitoring the execution of these programs through audit trail analysis. Auditing is the process of logging "interesting" activities. In current systems, auditing is carried out at the system call interface. Audit trails contain all system calls invoked, and hence, can be used to monitor the execution of privileged programs.

We describe a prototype program execution monitor running under the Sun Basic Security Module (BSM) Unix operating system. Figure 6 depicts the architecture of the execution monitor. The preprocessor filters the audit trails generated by the audit daemons, and associates audit records with the sub-

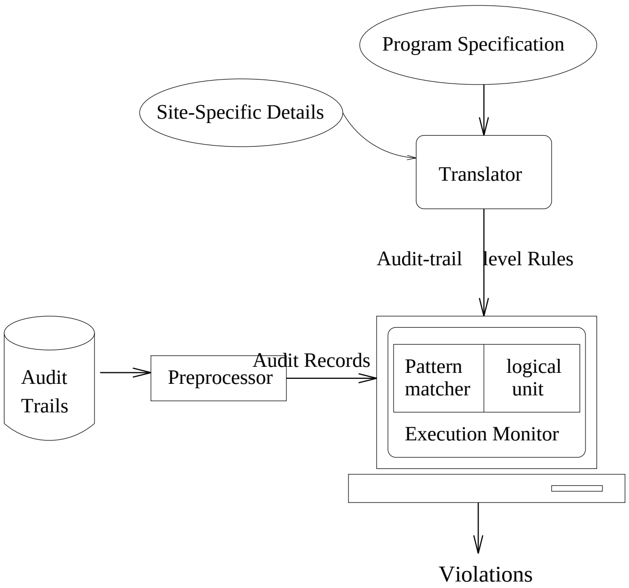

Figure 6: Architecture of the Execution Monitor

ject identifier (user, program). It generates internal audit records of the following form:

```
(uid, progid, op, euid, egid, path, pmode,
     ouid, ogid, devid, fileid, port)
```

`uid` and `progid` are the subject identifiers of the record; `op` is the operation; `euid`, `egid` are the effective user id and the effective group id of the process; `path`, `pmode`, `ouid`, `ogid`, `devid`, and `fileid` are the pathname, permission mode, owner uid, group uid, device id, and the file id of the file if the operation is on a file; `port` is the port number if the operation is on a port.

The translator is responsible for the automatic transformation of the program policy to audit-trail rules, which are logical expressions in terms of the fields of the internal audit record. Thus, we can match the operations of the subject with security specifications.

A practical problem with a Sun BSM audit trail is that audit records are not associated with a program. To associate all the records with the appropriate (user, program) identifier, we keep a table of the program each process is currently executing. As we are only interested in privileged programs, we treat all non-privileged programs as a single program. The preprocessor keeps track of all the exec and fork calls and updates the table in the following ways:

- exec: when a user running a non-privileged program invokes a setuid root program P, the program associated with the new process becomes P.
- fork: when a new process is created, the program associated with the new process is temporarily that of its parent, pending its invocation of a different program.

In the case of a server or a daemon, the situation is more subtle. A server/daemon is usually associated with a pseudo-user. However, sometimes a server/daemon may execute a command or run a program on behalf of a user. For instance, the cron daemon in Unix is actually a background daemon used to execute time-dependent jobs on behalf of users. To execute a command/program for a user, the server forks a new process, the new process changes its uid to that of the user, and then the new process executes the command or program for the user. Therefore, the preprocessor also tracks all the setuid/seteuid/setreuid calls made by server/daemon programs.

There are some limitations associated with audit trails. First, most auditing systems do not record all parameters to system calls. For instance, the record for a write call contains the file written to but not the data that was written. Still, current BSM audit trails are sufficient for monitoring a wide range of program behavior that bears on security. Second, a major benefit of our approach compared with conventional audit analysis in current IDS is that we do not need to analyze audit records associated with non-privileged programs. Therefore, only a small number of audit records is needed to be analyzed.

6 Example Program Policies

In this section, we revisit the programs described in Section 2 and describe the program policies of these programs. We discuss how the specifications can be used to detect exploitations of known vulnerabilities. Although the vulnerabilities are known, we specified the program policies solely based on the expected behavior of these programs. Therefore, we believe that our method can be effective in catching exploitations of yet-to-be-discovered vulnerabilities in privileged programs.

6.1 Finger Daemon

As mentioned in Section 2, the finger daemon program (versions before 11/5/88) has a vulnerability that enables an attacker to inject his own code and make the daemon execute the code. The vulnerability was exploited by the Internet Worm to execute a copy of the worm in hosts which provide the finger service. However, the intended behavior of the finger daemon is actually very restricted and easy to define as below:

```
PROGRAM fingerd(U)
   read(X) :- worldreadable(X);
   bind(79);
   write ("/etc/log");
   exec("/usr/ucb/finger");
END
```

Clearly, the worm attack is detected since fingerd is only allowed to execute the finger program. In addition, we believe anything an attacker can do to subvert the program with respect to security will violate the specification and will be caught by our execution monitor.

6.2 Rdist

In Section 2, we described informally the exploitation of one of the vulnerabilities of rdist. The following is the policy of the rdist program, including the behavior of both the client and server.

```
PROGRAM rdist(U)
   read(*);
   write(F) :- inside("/usr/home" + U.name, F.name);
   chmod(F) :- create(F);         (a)
   chown(F) :- create(F);         (b)
   bind(P) :- priv_port(P);
END
```

This specification simply says that although rdist is setuid root, and has the potential to do anything, it can write only to files that belong to the user who invokes it. In addition, rdist should change the ownership and the permission mode of the temporary file, i.e., it is allowed to change the ownership and the permission mode of the file it created (Rules a and b). This simple property can be checked easily at runtime using our execution monitor.

6.3 Sendmail

The following is the specficiation of Sendmail. Despite the numerous functions of this program (as indicated in Section 2), it still has a concise specification with respect to security. In general, Sendmail manipulates only files inside the mail spool directory and inside the mail queue directory. In addition, it may write to some configuration files in the /etc directory, where the file names are prefixed by "sendmail". One of the problems with Sendmail is that it can execute a predefined command for a user when new mail arrives. A typical example is the vacation program, which is used to automatically send a reply message. However, Sendmail is not intended to have the authority to execute any arbitrary program for a user. Thus, our policy defines what Sendmail can do for the superuser when new mail arrives. With this specification, an attacker who exploits Sendmail to perform any bad action (e.g., modifying the password file and system programs) will be detected.

```
#define mailboxdir          "/usr/spool/mail"
#define mailspooldir        "/usr/spool/mqueue"
#define mailport            25
#define root_mail_handler   "/home/root/mail_handler"
PROGRAM sendmail(U)
   read(X) :- worldreadable(X);
   write(X) :- inside(X, mailboxdir);
   write(X) :- inside(X, mailspooldir);
   write("/etc/sendmail."+"[\.]*");
   bind(mailport);
   exec("/bin/mail");
   exec(root_mail_handler) :- U.uid = 0;
END
```

7 Related Work

Similar approaches to specifying the access capabilities of programs have been proposed by Karger [12], King [13], and Lai [15]. Karger attempted to use a table-driven translation mechanism to deduce the set of files which will be accessed by a program (e.g., abc.obj, abc.exe for a fortran compiler). The file access policy is enforced with a name checking subsystem. Instead of using a file translation scheme, Lai proposed a data structure called Valid Access List (VAL) to hold the set of files accessible by an untrusted process. He describes how to extend the operating system kernel to enforce the file access policy. In [13], King used a regular-expression-based language to describe the set of objects each operation can access. Instead of a dynamic technique, static program analysis is used to check whether a program conforms to the policy. What distinguishes our work is we are trying to limit the damage caused by errors in privileged programs, while others are trying to limit the damage caused by a trojan horse or virus. We are more concerned about a normal user misusing the privileges of a privileged program, not a privileged program misusing the privileges of a user. In addition, we find that the behavior of privileged programs are more regular, so that the policies for these programs can be easily specified.

Apart from monitoring the execution of privileged programs, the approach of software testing is proposed to detect flaws in the implementation of privileged programs. Property-based testing [8, 9] employs security specifications (similar to our specifications for privileged programs) as the basis for automated static and dynamic testing of privileged programs from their source code. It uses the technique of slicing, dataflow coverage metrics, symbolic evaluation, and execution monitoring.

8 Conclusions and Future Work

In this paper, we discuss a method for detecting exploitations of vulnerabilities in privileged programs. Our method uses specifications to describe the intended behavior of privileged programs, and uses audit trails to monitor the actual behavior of the programs. We identify an important aspect of the behavior of privileged programs: the set of accesses. We describe a program policy specification language for specifying the security relevant behavior of privileged programs based on predicate logic and regular expressions. The language is capable of expressing a wide range of program behaviors. We presented the specifications of some "problematic" privileged programs in Unix, showing that previous exploitations of the vulnerabilities in these programs are indeed inconsistent with our specifications. In addition, the specifications are surprisingly short and concise, and the computation requirements of the audit trail analysis is low, making real-time detection possible. Our work can also be the basis for a misuse detector in an intrusion detection system, detecting "misuses" that exploit vulnerabilities in privileged programs. Conceptually, our approach uses detection as a mean to enforce the "least privilege principle" [11]. We believe it is the first attempt to detecting attacks by identifying deterministically the positive behavior of objects (programs). We find that our approach works well on most of the privileged program in Unix, except for some authentication programs (e.g., login, rlogin). Although our approach is not complete, we strongly believe that by appropriately restricting the behavior of privileged programs, the chance of security compromise due to errors in these programs can be greatly reduced.

For future work, we see an immediate application of our approach to intrusion detection systems (e.g., NIDES [16], DIDS [18]). In addition, our basic approach of detecting "misuses" of objects (in this case: privileged programs) by specifying the positive behavior should also be applied to other system components such as DNS, NFS, and routers. Next, to overcome the limitations of system audit trails, the approach of application auditing, i.e., in addition to the kernel, the application program itself generates audit trails is promising. Briefly, the goal is to find a way to instrument a program automatically to generate audit trails that provide information needed for monitoring. Application auditing depends on the trustworthiness of the audit trails, which system auditing can, in principle, provide. Last, one observation is that although the size of a privileged program is often very large, the part of the program which needs privileges is often small. Therefore, a methodology for writing a program in a way that particular properties of the program can be guaranteed by trusting only a part of the source code would be very useful.

References

[1] Private communications with tsutomu shimomura.

[2] B. So B. P. Miller, L. Fredriksen. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12), December 1990.

[3] S. M. Bellovin. There be dragons. *Proceedings of 1992 USENIX Security Symposium*, September 1992.

[4] B. Cheswick. An evening with berferd: In which a cracker is lured, endured, and studied. *Proceedings of the Winter USENIX Conference*, January 1992.

[5] D. Clark and D. Wilson. A comparision of commercial and military computer secruity policies. *Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy*, April 1987.

[6] D. Denning. An intrusion detection model. *Proceedings of the 1986 IEEE Symposium on Research in Security and Privacy*, pages 118–131, April 1986.

[7] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of novermber 1988. *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, April 1989.

[8] G. Fink, C. Ko, M. Archer, and K. Levitt. Toward a property-based testing enviornment with applications to security critical software. *Irvine Software Symposium*, 1994.

[9] G. Fink and K. Levitt. Property-based testing for security critical software. *Proceedings of the 10th Computer Security Application Conference*, 1994.

[10] K. Ilgun. A real-time intrusion detection system for unix. *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*, pages 16–28, May 1988.

[11] M. D. Schroeder J. D. Saltzer. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), March 1975.

[12] P. A. Karger. Limiting the damage potential of discretionary trojan horse. *Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy*, April 1987.

[13] M. M. King. Identifying and controlling undersirable program behaviors. *Proceedings of the 14th National Computer Security Conference*, 1992.

[14] C. Ko and K. Levitt. Specifying and monitoring privileged program behavior. Technical report, University of California, Davis, 1994. (in preparation).

[15] N. Lai and T. E. Gray. Strengthening discretionary access controls to inhibit trojan horses and computer viruses. *1988 USENIX Summer Symposium*, June 1988.

[16] T.F. Lunt, A. Tamaru, and F. Gilham. A real-time intrusion-detection expert system (ides). Technical Report Project 6784, SRI, Menlo Park, Feburary 1992.

[17] S.E. Smaha. Haystack: An intrusion detection system. *Proceedings of the 4th Computer Security Application Conference*, October 1988.

[18] S.R. Snapp and et. al. Dids - motivation, architecture, and an early prototype. *Proceedings of the 14th National Computer Security Conference*, pages 167–176, October 1991.

[19] E. H. Spafford. The internet worm program: An analysis. *ACM SIGCOM*, January 1989.

[20] Sun Microsystem. *Man Pages: Rdist - remote file distribution program.*

[21] Sun Microsystems. *Sun Security Bulletin #122 - 126.*

[22] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. *Proceedings of the 1989 IEEE Symposium on Research in Security and Privacy*, pages 280–289, May 1989.