An Approach to UNIX Security Logging[1]

Stefan Axelsson, Ulf Lindqvist, Ulf Gustafson, Erland Jonsson

Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
email: {sax, ulfl, ulfg, erland.jonsson}@ce.chalmers.se

Abstract

Off-line intrusion detection systems rely on logged data. However, the logging mechanism may be complicated and time-consuming and the amount of logged data tends to be very large. To counter these problems we suggest a very simple and cheap logging method, *light-weight logging*. It can be easily implemented on a Unix system, particularly on the Solaris operating system from Sun. It is based on logging every invocation of the *exec(2)* system call together with its arguments. We use data from realistic intrusion experiments to show the benefits of the proposed logging and in particular that this logging method consumes as little system resources as comparable methods, while still being more effective.

1 Introduction

The main problem with collecting audit data for a log is not that it is difficult to collect a enough data, but rather that it is altogether too easy to collect an overwhelming amount of it. The sheer volume of the audit data is the immediate reason that logging is often considered a costly security measure. The collection of a large amount of audit data places considerable strain on processing and storage facilities, not to mention the time that must be spent, either manually, or aided by computers, sifting through the logs in order to find any breaches of security. The trade-off is between logging too much, and being drowned in audit data, and logging too little to be able to ascertain whether indeed a breach has taken place [Ande80, Lunt93, Mukh94].

Most UNIX installations do not run any form of security logging software, mainly because the security logging facilities are expensive in terms of disk storage, processing time, and the cost associated with analysing the audit trail, either manually or by special software. In this paper we suggest a minimal logging policy, lightweight logging, based on the one single system call *exec(2)*. We use empirical data derived from practical intrusion experiments to compare the lightweight logging method with a few other simple methods. It is concluded that the *intrusion traceability* of the proposed logging method is superior to that of the comparable methods.

2 The purpose of logging

The main purpose of logging for security reasons is to be able to hold users of the system accountable for their actions [DoD85]. Logging is one of two basic requirements for this, the other being identification/authentication. It is impossible to hold a user accountable for some action indicated in the logs if it can not be excluded that someone else has ”masqueraded” as the user.

Even though less than perfect accountability may result from the mere existence of a log, the logging mechanism serves other useful purposes [NCSC88]:

- It makes it possible to review the patterns of use, of objects, of users, and of security mechanisms in the system and to evaluate the effectiveness of the latter.
- It allows the site security officer to discover repeated attempts by users of the system to bypass security mechanisms.
- It makes it possible for the site security officer to trail the use (or abuse) that may occur when a user assumes privileges greater than his or her normal ones. While this may not have come about as a result of a security violation, it is possible for the user to abuse his or her privileges in the new role.
- The knowledge that there is a mechanism that logs security relevant actions in the system acts as a deterrent to would-be intruders. Of course, for a security logging policy to be effective in a deterring capacity, it must be known to would-be intruders.

[1] Published in *Proccedings of the 21st National Information Systems Security Conference*, pp. 62-75, Oct. 5-8, Crystal City, Arlington, VA, USA, 1998

- The existence of a log makes ”after the fact” damage assessment and damage control easier and more effective. This in turn raises user assurance that attempts to bypass security mechanisms will be recorded and discovered. Logs are a vital aid in this aspect of contingency resolution [Kahn95].

In UNIX environments in general, and in the systems under discussion in particular, some of the above mentioned aims cannot be fully realized. For instance, once a user has assumed super-user privileges in a UNIX system, he (or she) then typically has the power to turn off logging, alter existing logs, or subvert the running logging mechanism to make it provide a false record of events. Furthermore, UNIX systems typically do not use sufficiently strong methods of authentication to make it possible to hold a user accountable on the grounds of what appears in an audit trail. In either case, the knowledge that a security violation has taken place is to be much preferred to the situation in which a breach of security has taken place, but gone unnoticed.

3 Lightweight logging

3.1 Definition

We strive for a logging policy that would allow us to detect and trace attacks against our system, i.e. that could be incorporated into an intrusion-detection system (IDS) and by its mere simplicity facilitate the postmortem intrusion-detection task. Our main purpose is to provide an audit trail from which the security officer can establish exactly what occurred, and how it occurred, rather than merely being able to detect that some sort of significant event has taken place. A logging policy should meet the following requirements:

(1) The system should be transparent to the user, i.e. it should behave in the manner to which he has been accustomed.

(2) Since system resources are always sparse, as little as possible should be consumed. This means minimizing the use of storage space, processing time, and time spent by the administrator.

(3) While meeting the above requirements, sufficient data should be recorded to maximize our chances to detect and trace any, and all, intrusions.[1]

We have found that it would be possible to trace *most* of the intrusions presented in this paper by logging relevant information about each *exec(2)* system call made in the system. Since the number of *exec(2)* calls roughly corresponds to the number of commands issued by the user, the amount of audit data should be in the same order as that of *pacct*,[2] while recording more security relevant data than *pacct* does.

Unfortunately one cannot configure the SunOS BSM audit mechanism (see Section 4.2) to generate one record for every command executed, like *pacct*. If one wishes to record every invocation of the *exec(2)* system call, one must audit all the system calls in that audit class, in total 15 different system calls. This may produce more audit data than we care to store and process. Furthermore, the arguments to the *exec(2)* call are not recorded, and that fact reduces the quality of the audit data considerably.[3]

3.2 Example

The example below is a classic UNIX intrusion scenario that can be exploited to gain super-user privileges. This security flaw was present in SunOS 4.1.2, the version on which the first experiment was conducted. In order for the flaw to exist, there must be a shell script somewhere on the system that is *setuid* or *setgid* to someone, i.e. it is run with the privileges of its owner, or group, not its caller. The flaw is exploited by the intruder calling the shell script via a symbolic link, and this results in the intruder gaining access to an interactive command interpreter, henceforth called shell.

Table 1: Penetration scenario.

Step	Shell Command	Comment
1	ln -s /u/vulnerable-file -i	Make link to a *setuid* root shell script.
2	-i	Invoke the shell script as -i.
3	root#	The user now has an interactive root shell.

[1] Our intrusion data were collected on the premise that the attackers operated as insiders. In order to log data relevant to tracing intrusions from outsiders, a network security tool such as *Tcp wrapper* could be combined with our suggested logging mechanism. See [Vene92] for a description of *Tcp wrapper.*

[2] See Section 4.2.2 for a more detailed presentation of *pacct*, the UNIX process accounting facility.

[3] Both these restrictions have been lifted in SunOS 5.x.

This flaw comes about as a result of a bug in the UNIX kernel. When the kernel executes the shell script, it first applies the *setuid* bit to the shell and then calls the shell with the filename of the shell script as the first argument. If this filename is *”-i,”* the shell mistakes this for the command line switch to start in interactive mode. In later versions of SunOS, 5.x this problem has been corrected.[1]

To analyse what needs to be recorded in order to trace this intrusion, we look at the system calls made when exploiting this flaw.

Table 2: System calls.

Step	System calls invoked	Comment
1	fork() execve(”/bin/ln”, ”ln”, ”/u/vul...”, ”-i”) stat (”-i”, 0x9048) = -1 ENOENT (No such file or directory) symlink (“/u/vulnerable-file”, “-i”) = 0 close (0) = 0 close (1) = 0 close (2) = 0 exit (0) = ?	make link to a *setuid* root shell script
2	fork() execve(”-i”, ”-i”, ...) sigblock (0x1) = 0 sigvec (1, 0xf7fff94c, 0xf7fff940) = 0 sigvec (1, 0xf7fff8d4, 0) = 0 sigsetmask (0) = 0x1 sigblock (0x1) = 0 .	Invoke the shell script as -i The shell starts with some calls to *sigblock sigvec* and *sigsetmask*. The system calls that are executed are then dependent on the input to the shell.
3	root#	

Steps 1) and 2) in Table 2 detail the system calls that are invoked when running *ln(1V)* and *sh(1)*. Both these commands are executed by a shell that performs the *fork(2V)*/*exec(2)* sequence, which is a prerequisite for all command execution in UNIX.

We outline our suggestion for what information to be included in the audit record in Table 3.

Table 3: The proposed system call logging.

Information recorded for *execve(2)*	Step 1 (ln) (example)	Step 2 (sh) (example)
a record creation time stamp	xxxx1	xxxx2
the real UID	5252	5252
log UID	YY	YY
effective UID	5252	0
real GID	11	11
effective GID	11	11
process ID	1278	1280
parent process ID	1277	1277
filename	“/bin/ln”	“./-i”
current working directory	/u/hack	/u/hack
root directory	/	/
return value	success	success
argument vector to *execve(2V)*	“-s”, “/u/vulnerable-file”, “-i”	“-i”

[1] The filename is no longer passed as the argument to the shell. Instead, the shell is passed a filename on the form */dev/fd/X* where *X* refers to the file descriptor of the already open file. See [Steve92] for an introduction to the */dev/fd* interface.

Perhaps the only field in Table 3 that merits further comment is the field ”log UID.” We propose that each user be assigned a unique identifier when he logs into the system. This identifier does not change for the duration of the session, even if the user’s real UID changes, as a result of an invocation of the command *su(1V)* for instance. The existence of the ”log UID” field makes it easier to trace the commands invoked by each user, although it is not strictly necessary. The same information may be distilled from complete knowledge about the branch on the process tree from the root (login) to the leaf (the current process). The log UID simplifies this task; we have borrowed the concept from C2 auditing [NCSC88, DoD85].

From the above audit records it becomes clear that user *5252* executed a *ln(1V)* command that made a soft link with the name ”*-i*” to the shell script, and that the user then invoked the shell script via the link.

If we look in detail at the above, it becomes clear that we need only log the invocations of the *execve(2V)* system call made by this user to trace the intrusion. Since we log the argument vector (argv) to the *execve(2V)* call, we need not log the symlink call separately. As can be seen above, that information is recorded when we log the argument vector to the *ln(1V)* command. We have all the data necessary to trace this specific intrusion back to the user that performed it.

In essence, the proposed logging scheme, creates one audit record per command issued. This also holds true for regular accounting, but there are several differences:

- By logging the start of execution of every command instead of the end of execution, we have a better chance of detecting an ongoing intrusion attempt. This is especially true if we consider long running commands that crack passwords or search the filesystem for example. Furthermore, the command that commences the intrusion is logged. This is far from certain if we delay logging until the command has completed execution, since this may already have turned off auditing etc.
- The most severe security intrusions in UNIX environments are often performed by tricking a *setuid* program into performing some illicit action. By logging both the real and effective UID every time a command is to be run, we can detect many such intrusions.
- Regular accounting logs the first eight characters of every finished command but, since programs can be copied and renamed, this is easy to circumvent. By logging the full path name of every command, together with all arguments, the proposed auditing policy is much more difficult to trick.

4 The logging during the data collection experiment

4.1 The experiment

During the years 1993-1996, we performed a number of intrusion experiments in UNIX systems [Jons97, Olov95]. The original goal of these experiments was quantitative modelling of operational security, that is, we tried to find measures for security that would reflect the system’s ”ability to resist attacks.” In order to do so, extensive logging and reporting were enforced and a great deal of data were generated. We believe that these data are also useful for the validation of the logging policy proposed in this paper.

During the experiments a number of students (about 25) were allowed to perform intrusions on a system in operational use for laboratory courses at the Department of Computer Engineering at Chalmers in Sweden. The system consisted of 24 SUN ELC disk-less workstations and a file server, all running SunOS 4.1.2 or SunOS 4.1.3_U1. The system was configured as delivered, with no special security enhancing features [Broc94].

The attackers, who worked in pairs, were given an account on the system - thus, they were ”insiders” - and were encouraged to perform as many intrusions as possible. Their activities were limited by a set of rules meant to avoid disturbing other users of the system and to ensure that the experiment was legal. Further details are found in the references above.

4.2 The logging

There were three main classes of accounting in the experiment system that were active:

Connect time accounting is performed by various programs that write records into */var/adm/wtmp*, and */etc/utmp* [Sun90]. Programs such as *login(1)* update the *wtmp(5V)* and *utmp(5V)* files so that we can keep track of who was logged into the system and when he was logged in.

Process accounting is performed by the system kernel. Upon termination of a process, one record per process is written to a file, in this case */var/adm/pacct*. Process accounting’s main purpose is to provide the operator of the system with command usage statistics on which to base service charges for use of the system [Sun90].

Error and administrative logging is primarily performed by the *syslogd(8)* daemon [Sun90]. Various system daemons, user programs, or the kernel log abnormal, noteworthy conditions via the *syslog(3)* function. These conditions end up in */var/log/syslog* file on the experiment system.

Another class of logging designed with security in mind is the SunOS BSM (Basic Security Module) logging sub system [Sun90]. This logging facility is said by Sun Microsystems to conform to the requirements laid forth in TCSEC C2, even though the SunOS BSM has not been formally certified according to TCSEC. This logging mechanism was not active in the experiment system.

The system logging and accounting files in the experiment system thus consist of */usr/adm/wtmp*, */etc/utmp*, */var/log/syslog*, */usr/adm/pacct*, and */var/adm/messages*.

4.2.1 Connect time accounting

Various system programs enter records in the */usr/adm/wtmp* and */etc/utmp* files when users log into or out of the system. The purpose of the *utmp(5)* file is to provide information about users currently logged into the system, and the entry for the particular user is cleared when he logs out of the system. The *wtmp(5V)* file is never modified in this manner; instead, when the user logs out, another entry is made containing the time he left the system. The *wtmp(5V)* file thus contains a record of each user as he entered and exited the system.

The *wtmp(5V)* file also contains information indicating when the system was shut down or rebooted and when the *date(1V)* command was used to change the system time.

The *wtmp(5V)* records contain the following information:

- The name of the terminal on which the user logged in.
- The name of the user who logged in.
- The name of the remote host from which the user logged in, if any.
- The time the user logged into or out of the system.

4.2.2 Process accounting by pacct

The process accounting system is, as mentioned before, designed to provide the operator of the system with command usage statistics on which to base service charges for use of the system. The *pacct* system is usually activated by the *accton(8)* when booting the system. When active, the UNIX kernel appends an audit record to the end of the log file, typically */usr/adm/pacct*, on the termination of every process. The audit record contains the following fields:

- Accounting flags; contains information indicating whether *execve(2V)* was ever accomplished and whether the process ever had super-user privileges.
- Exit status.
- Accounting user.
- Accounting group ID.
- Controlling terminal.
- Time of invocation.
- Time spent in user state.
- Time spent in system state.
- Total elapsed time.
- Average memory usage.
- Number of characters transferred.
- Blocks read or written.
- Accounting command name; only the last eight characters of the filename are recorded.

4.2.3 Error and administrative logging

Beside the functions described above, many user and system programs use the logging facility provided by the *syslog* service. At system start-up, the logging daemon *syslogd(8)* is started, and processes can then communicate with *syslogd(8)* via the *syslog(3)* interface.

The messages sent to *syslog(3)* contain a priority argument encoded as a *facility* and a *level* to indicate which entity within the system generated the log entry and the severity of the event that triggered the entry. The *syslog* service is configured to act on the different *facilities* and *levels* by appending the message to the appropriate file, write the message on the system console, notify the system administrator, or send the message via the network to a *syslogd(8)* daemon on another host.

In the experiment system, *syslog* was configured to append all messages to */var/adm/messages* and debug messages from *sendmail(8)* to */var/log/syslog*.

5 Evaluation of the intrusion data with respect to different logging methods

During our experiments we defined an intrusion as the successful performance of an action that the user was not normally allowed to perform. About 65 intrusions were made, most of them already known by the security community, e.g., by CERT.[1] However, while CERT only informs about what system vulnerability was used for a specific intrusion, our experiment yielded further data. The greatest advantage of our intrusion data is that we know *exactly how* the intrusion was performed, which obviously is of specific interest when discussing logging for intrusion-detection purposes. Therefore, we categorize our intrusions according to what kind of audit trail they leave. For each intrusion class, we discuss the possibility of detecting an attack with normal system accounting or monitoring, and by means of using the suggested lightweight logging method.

The intrusions are categorized in ten broad classes as presented in the rest of this section. For the sake of brevity, only one typical intrusion in each class is described in detail, while the others are outlined. The discussion is structured under the following headings:

System logging; the logging performed by the kernel and system processes such as *init(8),* and *pacct(8).*

Application program logging; the logging performed by *application programs*, such as *su(1V)* etc.

Monitoring resource utilization; some attacks result in abnormal load on CPU, disk, network, etc., and this can be monitored, and anomalous behaviour can be detected. In practice, many intrusions are detected because the users of the system report that it acts ”funny” in this respect.

Lightweight logging (execve(2V)); discussion of the validity of the recorded information related to the suggested logging policy.

5.1 Class C1: Misuse of security-enhancing packages

There are several programs available that help the supervisor of a UNIX system to increase security by testing for known security problems. These programs can of course also be (ab)used by an attacker to learn about existing flaws in the attacked system.

Crack The target system did not enforce password shadowing, so any user was able to read out the encrypted password values and mount a dictionary attack by obtaining and executing the publicly available password guessing program *Crack*.

System logging: The execution of these kinds of programs, which are well-known packages consisting of several subprograms, leaves distinct patterns of multiple entries in the *pacct* file that are easy enough to detect and trace, provided the commands are not renamed.

Application program logging: N/A.

Resource utilization: Possibly massive disk (network if NFS) and CPU utilization.

Suggested logging: The program is recorded and saved with its arguments. Each spawned subprogram in the collection will also be recorded together with its arguments. This approach gives more accurate information considering the patterns in the log file.

COPS Intended to be used by system administrators to find security problems in their UNIX installations, *COPS* is a publicly available package that can also be used by attackers. It consists of a set of programs, each of which tries to find and point out potential security vulnerabilities.

Password generation rules When assigning passwords to other students attending diverse courses at the department, a program that randomly creates 7-character lower-case passwords is used. To make the passwords pronounceable and thus easier to memorize, the program makes every password contain 3 vowels and 4 consonants in a distinct pattern. Unfortunately, it turns out that this pattern severely limits the randomness of user passwords and makes exhaustive search feasible. The attackers compiled a dictionary that satisfied the password generation rules and then ran *Crack*.

Conclusion: Both system logging and our proposed logging policy are capable of detecting the use of the security-enhancing packages encountered during the experiment.

[1] See *http://www.cert.org* for information about CERT, and the advisories they publish.

5.2 Class C2: Search for files with misconfigured permissions or *setuid* programs

A general search of the filesystem for files for which the attacker has write permission, or files that are *setuid* to some user is often a step performed by the security packages mentioned in C1. We list it as a separate class since many of our attackers performed such a search when first trying to breach security. These attacks also have in common that they are resource-intensive in terms of (network) disk traffic and may be detected because of this.

Search for files with public write permission The target in this attack was carelessly configured permissions on diverse files in the system, especially system files or user configuration files. Files with public write permissions can be modified by arbitrary users, compromising the integrity of the system. During the experiment we chose not to count general searching as a breach in itself; to regarded the action as a breach, we demanded a detailed description of how to exploit the vulnerable file.

System logging: If the *find(1)* command is used, traces of that can be found in *pacct*. However, since the arguments to the *find(1)* command are not available, it is doubtful whether the security administrator can tell the difference between benign uses of *find(1)* and uses that are consistent with an ongoing intrusion attempt.

Application program logging: N/A.

Resource utilization: Possibly massive disk (network if NFS) and CPU utilization.

Suggested logging: The arguments to *find(1),* together with the command name, are logged making it possible to discover what the attacker is searching for. If the attacker tries to hide the name by making soft or hard links to the find command, the links are also traceable.

Search for *setuid* files Wrongly configured *setuid* files may compromise overall system security, especially *setuid* shell scripts or *setuid* programs with built-in shell escapes. In this case we also demanded a detailed description of how to exploit the vulnerable file.

Conclusion: The suggested logging policy can detect usage of *find(1)* for purposes of searching for files as above. System logging, on the other hand, has a difficult time differentiating between suspect and legitimate uses of *find(1)*.

5.3 Class C3: Attacks during system initialization

As the experiment system was configured it was possible to attack it by halting it during system initialization. The single-user root privileges thus obtained were able to be further exploited to become multi-user root.

Single user boot It was possible to boot the clients from the console to single-user mode. This was possible because */etc/ttytab* was not set up to secure console login. Hence, it was possible to modify an arbitrary filesystem on the client. (Although the clients were disk-less, their root directories were mounted via NFS from the file server.)

System logging: Through */var/adm/wtmp*, */var/adm/pacct* and */var/adm/messages* it is possible to tell when the machine was rebooted after it has come up in multi-user mode again. However, the commands executed in single-user mode are not logged, since logging is not active in single-user mode.

Application program logging: N/A.

Resource utilization: limited.

Suggested logging: It is possible to log the actions performed during single-user mode, but this is not normally done. The single-user mode is viewed as a transient administrative state, employed for administrative duties or system repair. As such, the resources necessary for running the logging mechanism may be unavailable. In our case, all but the root partition was mounted read only, making it difficult to store the log on disk.

Inserting a new account into the /etc/passwd file This is primarily a method to become multi-user root. After becoming single-user root, it is possible to insert a new account in the password file. It is then possible to log into that account when the client comes up in multi-user mode.

***Setuid* command interpreter/program in root filesystem** As single-user root, it is possible to make a copy of a command interpreter (shell), change the owner of the copy to an arbitrary user (for example root) and set its *setuid* flag. Then, when the host has entered multi-user mode, the attacker need only execute the copied shell to take the identity of its assigned owner. This method was primarily used to become multi-user root.

File server intrusion through *setuid* program Although the clients were disk-less, all modifications on the clients root filesystems were also available to the users on the file server. In this way, it was possible to become another user by executing a *setuid* program as described in the preceding intrusion scenario.

Conclusion: Since the system is not fully operational during initialization, it is difficult for both methods to detect, let alone trace, any intrusion attempt. It would be technically difficult to design a logging mechanism that would function under such circumstances, since we are in effect giving away super-user privileges to anyone who happens to walk by. This turns this attack into an ”outsider” attack, and we must monitor physical access to the computer room in order to have a chance of catching the intruder.

5.4 Class C4: Exploiting inadvertent read/write permissions of system files

As the experiment system was configured, many critical system files and directories were set up with inadvertently lax access permissions. This made it possible for the attackers to modify critical files to subvert system programs, or, in one case, to read data to which the attackers should not have had access.

YP configuration error NIS (see *yp*(*3R*))was installed according to the manual, meaning that it was necessary to initialize */var/yp* before bringing the machine up to multi-user mode. Doing this, and neglecting that an appropriate *umask* is not activated by default in single-user mode, may result in dangerous file permission settings on files created under those circumstances. The NIS server configuration database, */var/yp*, in the target system had permission mode 777, that is, writable and readable for all.

/etc/utmp writable The default on the experiment system was for */etc/utmp* to be writable for all users. This makes it possible for an intruder to hide from appearing in the output of commands such as *who(1)* and *users(1),* but, more importantly, it allows a user to alter system files. This is accomplished by editing */etc/utmp* and then issuing a *setuid* command (*write(1)* for instance) that uses */etc/utmp* to find its output file.

System logging: *pacct* will show only that some user issued, for instance, a *write(1)* command that looks benign enough. There is certainly no way of differentiating this use of *write(1)* from ordinary legitimate uses of that command.

Application program logging: N/A.

Resource utilization: Limited.

Suggested logging: Unfortunately, if the command the user uses to manipulate */etc/utmp* does not stand out in the audit trail, this kind of intrusion will be difficult to trace. Since we log all arguments to commands issued, our chances of catching the modifying command will have increased substantially in comparison with the chances that *pacct* has.

Crash the X-server When a client wishes to connect to the X-server on the local machine, the client looks for a UNIX domain socket at a predefined location in the filesystem (*/tmp/.X11-unix/X0* in the experiment system). This socket and its directory were world-writable in the experiment system and, as a result, the user could remove the socket and thus hang the X-server. By replacing the socket with a non-empty directory and forcing the X-server to restart, the directory is unlinked, and the files are left ”hanging.” This would require the supervisor to manually fix the system on the next boot with *fsck(8).*

Reading of ”mounted” backup tapes In the experiment system, backup tapes were present in the tape streamer awaiting that night’s backup run. Since the backup tapes were constantly reused (a fairly common policy in many installations) and the tape streamer’s device file had world-read permissions set, it was possible for the attackers to read the previous week’s backup tapes. This enabled them to read files that they would not normally be allowed to read.

Conclusion: Misuse of system files with erroneous permissions is often detectable by the proposed logging scheme. This misuse often manifests itself as a suspect argument to a user command, that is, the command accesses a file that it should not normally access. System logging has very slim chances of detecting this, since arguments are not logged.

5.5 Class C5: Intercepting data

Due both to configuration errors and the nature of certain UNIX system applications, it was possible for the attackers to intercept communication between users and the experiment system.

Snooping the X-server A publicly available program called *xkey* was used to listen to traffic to the X-server. This program tries to connect to the X-server on the target machine and, if the request is granted, makes it possible to intercept any keystrokes typed by the console user at the target machine.

Frame buffer grabber If a person can log into a SunOS 4.1.x workstation from a remote host, he or she may be able to read the contents of the console’s video RAM memory. The frame buffer character special file, */dev/fb,* is per default writable and readable for everyone.

Ethernet snooping Many typical UNIX clients for remote login transmit authentication information in the clear via the network. It is thus possible for someone with access to the network (network topology and technology permitting) to eavesdrop on this traffic and learn passwords, etc. In most UNIX installations one must already have local super-user privileges to be able to perform this kind of attack, and this was the case in the experiment system when the attackers performed the intrusions.

System logging: *pacct* records the commands issued by every user, including root. If the super user has run any of the more popular network listening tools, they should appear in the log entries of *pacct*, that is, unless the intruder has already turned off logging, which unfortunately is very likely.

Application program logging: N/A.

Resource utilization: Listening to all network traffic (by setting the network interface in “promiscuous mode”) can load the local host heavily.

Suggested logging: The same argument as for *pacct* above applies. However, since the arguments to, for instance *tcpdump,* are recorded, it should be easier to differ between legitimate and illegitimate uses of *tcpdump*.

Conclusion: The snooping that is described above is performed by special programs, most of which are not part of the system distribution. It is easy for the attacker to rename popular packages when installing them, thus foiling our logging effort. In the experiment, most attackers did not bother with this, and hence their intrusions were relatively easy to trace, since these are programs that normally should not be run.

5.6 Class C6: Trojan horses

Some attackers made unsuspecting users execute Trojan horses, that is, applications that purported to do something benign, but did something sinister in addition to that.

Trojan su It is possible for anyone to create a fake *su(1V)* program that when executed saves a copy of the entered password and then prints an error message ”su: Sorry” as though the password were wrong. The program then erases itself from the filesystem.

System logging: Normally, when *su(1V)* is executed but the result is unsuccessful, a record starting with “*#su”* is found in *pacct*. The ”*#*” indicates that the program was executed with root privileges.[1] Consequently, the Trojan horse *su*, which does not run with root privileges, should appear as a plain “*su,”* which can easily be detected. We cannot judge whether this can be circumvented by a more carefully designed Trojan horse. With regard to tracing, the only thing *pacct* tells us is the user who executed the Trojan *su*, not the location or creator of the program.

Application program logging: N/A.

Resource utilization: limited.

Suggested logging: Enough information to trace this intrusion is recorded since the intrusion method requires that the command is invoked as *su*. Information is recorded on which user executed the program and the full path of the program, and both real UID and *setuid* settings are logged. Since the full path of the program is logged, the chances of discovering who actually planted the fake *su* program increases.

Trojan e-mail attachment This is somewhat of a social engineering attack. One group of attackers sent an e-mail message that contained what was announced to be a picture of explicit nature. However, the picture could not be viewed by normal software already installed on the system, but only with the supplied software, which was also attached to the e-mail. That software was a Trojan horse that hijacked the viewer’s account before it correctly displayed the picture.

Conclusion: System logging records only the last part of the path to the command. Because of this it is impossible to differentiate between the running of legitimate commands and their Trojan counterpart. The suggested logging policy detects these attacks, since it will show the execution of a normal system command with a suspect path, or the running of a command that has been introduced into the system in a suspicious way.

5.7 Class C7: Forged mail and news

Owing to the design of the mail and news servers on the experiment system, it was fairly easy to send a message that purported to be from someone else.

[1] However, the documentation is unclear as to the exact circumstances under which the ”*#*” is inserted by *pacct(2).* We have found that it is a fairly unreliable indicator as to what privileges were actually acquired by the executing program.

Faking email On many UNIX systems, it is possible to fake the sender of an e-mail message, making it appear to originate from another user or even from a non-existing user. This is done by connecting to the mail port through *TCP/IP* and interacting directly with the *sendmail(8)* daemon.

System logging: If the *telnet(1C)* command has been used, we can find a record of that in *pacct* on the sending machine and connect that to the records in *syslog(3)* (described below) through the time stamps.

Application program logging: If the message has been sent from one of our workstations, we can find tracks in */var/log/syslog* left by *sendmail(8)* via *syslogd(8)* on that machine, but only with the faked sender identity.

Resource utilization: limited.

Suggested logging: Clearly, not enough information is recorded to trace the sending of a fake e-mail. Partial information may be available depending on the way in which way the *telnet(1C)* command is invoked or whether the command *mconnect(8)* is used.

Forged news As a consequence of the way in which the remote USENET News server protocol was designed, it is fairly easy to forge a USENET News article to make it appear as though it originated from another user on the system. All authentication must be performed in the news-client software, but nothing prevents a user from connecting to a remote News server by hand, that is, by using *telnet(1C)*.

Conclusion: All attackers that forged mail and news did indeed use the *telnet(1C)* command, passing the parameters on the command line, and were thus easy to detect by the suggested logging, even if the actual mail or news article would be difficult to trace. However, it is trivial to invoke the *telnet(1C)* command without any command line arguments and foil both methods of logging.

5.8 Class C8: Subverting *setuid* root applications into reading or writing system files

Setuid root applications on UNIX systems are allowed to read or write to any file by default. It is imperative that such applications check user supplied arguments carefully, lest they be tricked into doing something that the user would not normally be allowed to do. However, many such applications contain flaws that allow an attacker to perform such unauthorized reading or writing of critical files.

Xterm logfile bug, version 1 The *xterm(1)* client has an option for the entire session to be logged to a file. Furthermore, *xterm(1)* is *setuid* root for it to be able to change the owner of its device file to that of the current user. A bug makes it possible for any user to specify an existing file as the logfile to *xterm(1)* and have *xterm(1)* append data supplied by the attacker to that file.

System logging: From the *pacct* file, all we can see is that someone has run *xterm(1)*, a so common occurrence that we can safely say that it is impossible to trace an intrusion this way.

Application program logging: N/A

Resource utilization: Limited.

Suggested logging: Since we log all the arguments to *xterm(1)* as it is being run, we catch both the invocation of the logfile mechanism and the telltale argument *-e echo ”roott::0:1::/bin/sh”* or a similar one, which is the hallmark of this intrusion.

Xterm logfile bug, version 2 A variation of the preceding exploit, where the attacker can create a file and have the output from *xterm(1)* inserted in that file, provided that the file does not already exist.

Change files through mail alias The UNIX operating system maintains a global mail aliases database used by the *sendmail(8)* program to reroute electronic mail. One standard alias delivered with some versions of UNIX is *decode*. By allowing this alias, it is possible for anyone to modify some directories in the system.

Finger daemon The *finger(1)* utility in the experiment system is *setuid* to root, and it makes an insufficient access check when returning information about a user. It is possible to make finger display the contents of any file via the use of a strategic symbolic link from *.plan* in the user’s home directory, to the target file.

Ex/vi-preserve changes file The venerable UNIX text editors *ex(1)* and *vi(1)* have a feature by which in the event that the computer crashes or the user is unexpectedly logged out, the system preserves the file the user was last editing. The utility that accomplishes this, *expreserve(8)*, is set UID to root, and has a weakness by which the attacker can replace and change the owner of any file on the system.

/dev/audio denial of service Owing to a kernel bug, it is possible to crash the machine by sending a file via *rcp(1C)* to */dev/audio* on a remote machine.

Interactive *setuid* shell This problem in SunOS 4.1.x has been fixed in more recent operating systems. An attacker simply invokes a *setuid* (or *setgid* or both) shell script through a symbolic link, which immediately results in an interactive shell with the effective UID (and/or GID) of the owner of the shell script. This is detailed in Section 3.2.

Conclusion: The methods used to trick the *setuid* program inevitably either supply suspect arguments to the command or modify the filesystem in advance to running the *setuid* command. Depending on the specific circumstances, our logging proposal has a high chance of detecting and tracing the intrusion, since we either log the suspect arguments themselves or log the commands that poison the filesystem. System logging does not manage to accomplish either.

5.9 Class C9: Buffer overrun

As mentioned above, a *setuid* program must be careful in checking its arguments. There exists a class of security flaws where the attacker subverts the *setuid* application by filling an internal (argument) buffer so that it overflows into the *setuid* program's execution context. In this way it is possible for the attacker to force the *setuid* program to execute arbitrary instructions. We have encountered only one such attack. We include it here because of the severity of this type of attack and because it is widespread and commonly encountered in the field.

Buffer overrun in rdist The *rdist(1)* utility has a fixed length buffer that can be filled and thus made to overflow onto the stack of *rdist(1),* which is set UID root. The attackers did not manage to exploit this other than to crash *rdist(1),* but we include it here in any case, as it's an interesting class of intrusions.

System logging: As usual, *pacct* does not manage to leave any conclusive traces in the log files. The invocation of the program with the overflow condition would probably not show up, since the name of the finished program is recorded at the end of execution when the original program image has typically already been overlaid with that of a set UID shell.

Application program logging: N/A.

Resource utilization: Limited.

Suggested logging: Since we log the arguments to the command, *rdist(1)* in this case, we immediately see that something is wrong. We in fact record in the log the entire program that *rdist(1)* is lured into overflowing onto its stack! (We may not always be so fortunate; some variations of these exploits keep the actual overflow code in an environment variable, which we will not log.) Since we see both the original *exec(2)* of *rdist(1),* followed by the *exec(2)* of a root shell, without the intervening *fork(2V),* we have conclusive proof that the system has been subverted.

Conclusion: See the discussion concerning the above exploit, since it is typical of these kinds of exploits.

5.10 Class C10: Execution of prepacked exploit scripts

It is possible for the novice attacker to download prepacked programs or command scripts that exploit a known security flaw. Such packages are in wide circulation and provide the attacker with an easy entry into the system.

Loadmodule *setuid* root script We have included a specific example of a breach that involves a *setuid* script since this *setuid* script is included in the SunOS distribution tapes. Hence, this security flaw is widespread and, as such exploit scripts have been developed, and widely circulated. This particular exploit script (*load.root*) was found by many experimenters who used it successfully, some without any deeper understanding of the security flaw involved. The exploit script in question uses the environment variable *IFS* to trick *loadmodule(8)* into producing an interactive shell with super-user privileges.

System logging: Since *pacct* does not record the arguments to the commands that the user executes, it is difficult to establish that the *load.root* exploit script has indeed been run. However, as the exploit script executes a number of commands in a predefined sequence, it should, at least in theory, be possible to ascertain that the script has been run with some level of certainty. This is generally not possible without detailed knowledge about the script. As mentioned previously, the ”#” marker to indicate that super-user privileges has been used does not appear in this case.

Application program logging: N/A.

Resource utilization: Limited.

Suggested logging: Since we can determine what commands were run and with what arguments, it becomes much easier to determine that a breach has occurred. In particular, the fact that the user has executed a script that in turn invokes *exec(”/bin/sh”, ”sh”, ”-i”)* with root privileges gives the game away. However, the flaw is exploited by the introduction of an environment variable, something which we do not record because this would lead to the storage of too much data. A strong indication that some sort of IFS manipulation has taken place, however, is the fact that the audit trail shows that a user has executed a command named *bin* as part of a shell script.

Mailrace (sendmail) The original mail handling client */bin/mail(1)* is set UID root in SunOS. If a recipient of a mail message lacks a mail box, the program creates one before it appends the mail message to it. Unfortunately, there exists a race condition between the creation of the mail box and the opening of it for writing. Exploit scripts have been published that exploit this race condition. These operate by replacing the newly created mail box with a symbolic link to any file which, as a result, will be overwritten or created with the contents specified by the attacker.

Conclusion: If the script is known beforehand, its use can be established with normal system logging. The suggested logging makes it a good deal more likely that a script that has not been previously examined can be traced and analysed to determine its effects.

6 Summary and discussion of results

After studying the above security breaches it becomes clear that the main system logging mechanism, *pacct,* suffers from three major shortcomings:

(1) It does not commit the executed command to the audit trail until it has finished executing. This misses crucial long running commands. Commands, the process image of which is overlaid by a call to *exec(2),* does not appear in the audit trail, and the command that crashes or compromises the system is at risk of not being included in the audit trail.

(2) *Pacct* does not record the arguments to issued commands. This more often than not turns the log material into a useless alphabet soup from a security perspective, since it is impossible to see what executed commands were set to act upon in terms of files etc. In most of the cases above, the arguments to the commands are what set legitimate uses of the commands apart from illegitimate ones.

(3) The mechanism that is supposed to trace uses of super-user privileges is unreliable at best and downright erroneous at worst. It can thus not be trusted to provide worthwhile information about the use of super-user privileges.

In Section 5 we have compared lightweight logging to three other logging methods: *system logging, application program logging,* and *monitoring resource utilization*. In order to make our comparison, we group all the these logging methods under the heading of *traditional* logging, since we believe that they would most often be employed together. The results are summarised in Table 4, which contains a total of 30 different attacks in 10 classes. We see that lightweight logging detects 21 intrusions in 7 classes, whereas traditional logging only covers 7 intrusions in 3 classes. The “coverage” - loosely defined as related to the number of missed intrusions - is for lightweight logging, about a factor 2.5 better than for traditional logging, e.g., 9 missed intrusions compared to 23. This is despite the fact that traditional logging collects audit data from more sources. However, neither method succeeded in detecting any of the attacks in the three classes C3, C5, and C7 (altogether 9).

Table 4: Summary of logging mechanism evaluation.

Class (number of intrusions in class)	Lightweight logging	Traditional logging
C1: Misuse of security enhancing packages (3)	X	X
C2: Search for files with misconfigured permissions or *setuid* programs (2)	X	
C3: Attacks during system initialization (4)		
C4: Exploiting inadvertent read/write permissions of system files (4)	X	
C5: Intercepting data (3)		
C6: Trojan horses (2)	X	X
C7: Forged mail and news (2)		
C8: Subverting *setuid* root applications into reading or writing system files (7)	X	
C9: Buffer overrun (1)	X	
C10: Execution of prepacked exploit scripts (2)	X	X
Number of classes:	7	3
Number of intrusions (30):	21	7

From this comparison it becomes clear that by correcting the three shortcomings in the original *pacct* mechanism, mentioned above, our proposed policy manages to trace an overwhelming portion of the intrusions, namely those that fall into the following three categories:

(1) The user has run commands he should not run; the log shows that someone with a log-UID that does not correspond to a known supervisor has executed commands with super-user privileges.

(2) The user has run commands with suspect arguments and, by doing so, has managed to trick a system application into doing something illicit.

(3) The user has run a suspicious-looking sequence of commands. This indicates that the user has run some exploit script or some security enhancing package that he should not have run.

Furthermore, as a result of this, our logging policy produces an audit trail from which more detailed information can be extracted, namely exactly *how* an intrusion was performed and not only that is *was indeed* performed. Thus, by also logging more security-relevant information pertaining to who executed the command, and the general environment in which it was executed, we have a sufficiently complete record of events on which we could base further action, provided that the audit trail is protected from manipulation. This can be accomplished by logging to a dedicated network loghost, or to a write-only media, as detailed in [Garf96].

7 Conclusion

We have shown that the lightweight logging method is more effective in tracing intrusion than comparable methods and that it traces an overwhelming majority of intrusions encountered during our experiments. It can very easily be implemented using the SunOS BSM module in newer versions of the SunOS operating system [Sun95]. Since it does not consume much resources in terms of processing power and storage capacity, it can be left running on all machines in an installation. Thus, it can be used as a “poor man’s logging.”

8 References

[Ande80] James P. Anderson. ”Computer Security Threat Monitoring and Surveillance.” *Technical report*, Box 42 Fort Washington, Pa. 19034, (215) 646-4706, February 26, 1980, revised April 15, 1980

[Bish95] Matt Bishop. ”A Standard Audit Trail Format.” *In Proceedings of the 18th National Information Systems Security Conference*, pages 136-145, Baltimore, Maryland, USA, October 10-13, 1995.

[Broc94] Sarah Brocklehurst, Bev Littlewood, Tomas Olovsson, and Erland Jonsson. ”On measurement of operational security.” In *Proceedings of the Ninth Annual Conference on Computer Assurance* (*COMPASS ’94*), pages 257-266, Gaithersburg, Maryland, USA, June 27-July 1, 1994.

[DoD85] U.S. Department of Defense. Trusted Computer System Evaluation Criteria, December 1985. DoD 5200.28-STD.

[Garf96] Simson Garfinkel and Gene Spafford. ”Practical UNIX & Internet” Security. O’Reilly & Associates, second edition, 1996.

[Jons97] Erland Jonsson and Tomas Olovsson. ”A Quantitative Model of the Security Intrusion Process Based on Attacker behaviour.” In *IEEE Transactions on Software Engineering*, vol. 23, No. 4, April 1997.

[Kahn95] Jay J. Kahn and Marshall D. Abrams. ”Contingency Planning: What to do when bad things happen to good systems.” In *Proceedings of the 18th National Information Systems Security Conference*, pages 470-479, Baltimore, Maryland, USA, October 10-13, 1995.

[Lunt93] Teresa F. Lunt, ”A Survey of Intrusion Detection Techniques.”, *Computers & Security*, 12(4):405-418, June 1993.

[Mukh94] Biswanath Mukherjee, L. Todd Heberlein, Karl N. Levitt, ”Network Intrusion Detection.” *IEEE Network*, May/June 1994

[NCSC88] National Computer Security Center. ”A Guide to Understanding Audit in Trusted Systems.” 1 June 1988. NCSC-TG-001, Version-2.

[Olov95] Tomas Olovsson, Erland Jonsson, Sarah Brocklehurst, and Bev Littlewood. ”Towards operational measures of computer security: Experimentation and modelling.” In Brian Randell et al., editors, *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series, chapter VIII. Springer-Verlag, 1995.

[Steve92] W. Richard Stevens. ”Advanced Programming in the UNIX Environment.” Addison-Wesley Publishing Company Inc., 1992, ISBN 0-201-56317-7.

[Sun90] Sun Microsystems. System and Network Administration, March 27, 1990. Part No: 800-4764-10, Revision A.

[Sun95] Sun Microsystems Inc. ”SunSHIELD Basic Security Module Guide.” Sun Microsystems Inc. 2550 Garcia Avenue, Mountain View, California 94943- 1100 USA.

[Vene92] Wietse Venema. ”TCP WRAPPER: Network monitoring, access control and booby traps.” In *Proceedings of the 3rd USENIX UNIX Security Symposium*, pages 85-92, Baltimore, Maryland, USA, September 14-17, 1992. USENIX Association.