RANDOLPH LANGLEY

auditing superuser usage

Randolph Langley is a member of the Computer Science Department at Florida State University. Prior to this, he worked both in the financial industry and for the Supercomputer Computations Research Institute.

langley@cs.fsu.edu

IMAGINE BEING THE MANAGER OF A UNIX group who, after receiving a telephone call that a user cannot access his NFS home directory, happens to find the following lines in the shell history file for the root account:

```
ps -elf | grep -i portmap
kill -TERM 2193
portmap -dlv
```

It appears that somebody was trying to debug the portmapper, but when was this done? Who did it? Is it the cause of the current problem, or was it someone working on this problem?

While in this case it might be merely desirable to know more about these lines—after all, you can just do a ps to find out if the portmap program is running and start it if it is not—it is sometimes necessary to maintain records of who does what on some production systems. Programs such as sudo [1] and op [2] provide a means of controlling the who and, to a lesser degree, the what, but determining more exactly what was actually done can still be a challenge.

One method of meeting this challenge, sudoscript [3], was presented in Howard Owen's August 2002 *;login:* article "The Problem of PORCMOLSULB." It wraps the execution of a shell by sudo [1] with a script session.[1] While this is certainly a viable approach, modifying script seemed to me the more natural approach.[2] I wanted to add a remote logging capability since this allows one both to centralize logging and to provide some fraction more capability in the event of a break-in via sudo (although certainly a knowledgeable cracker should be able to stop this logging quite quickly). Modifying script seemed the most direct way to provide such logging.

Rationale

In large organizations, the responsibility for system security and its monitoring has natural divisions: system administrators, their managers, the computer security group, and technology auditing all have different roles in preserving and monitoring system security. Division of responsibility also helps maintain accountability in the overall system. To divide responsibility, information technology controls should exist at every level, eliminating any single point of trust.

1. I would like to properly credit the program script to someone, but my detective skills have not sufficed to find the original author.
2. Just to make the sequence of events clear, I had done the main modifications to script before I was aware of sudoscript.

Traditionally, however, with UNIX system administration there has been an imbalance in accountability for superuser activities by system administrators. While important advances such as SELinux [4] introduce new and useful capability in the form of mandatory access controls in imposing limits, in an audit or forensic situation, tracking superuser actions typically has meant following whatever logs were available from shells and from what can be inferred from reading various system logs. Shell logs typically are not configured to keep timestamps (though many shells, such as bash [5], do have that option). Shell logs keep a record, not of the actual keystrokes, but, rather,of the command line that was eventually entered; shell logs do not keep track of the output from commands; they don't have the ability to automatically forward information to other machines designed to maintain security information.

While a machine such as a honeypot may have a designed-in system for fine-grained tracking of user interaction at a very low level, such as honeynet's use of sebek [6]—typically as a hidden kernel module, since such logging should not be obvious to the intruder—such modifications are not desirable in a typical production system.

Although the program sudo is commonly used in order to improve accountability, it also provides other benefits, such as limiting the number of people who need direct access to a superuser password. In addition, it provides some measure of limiting use of privilege by providing a means of allowing certain programs to be executed by a given user.

From a management perspective, simply knowing *who* did *what* can be invaluable, such as when tracking down ad hoc changes that were made in the heat of problem resolution but were not put into the boot-time configuration. For a technology auditor, superior tracking of superuser privilege allows the auditors to have a more informed opinion of operations. For a security officer who may be looking through the logs for security lapses, having better and more accurate logs of actions by superusers may be desirable.

Changes to script.c

While quite a bit of this can be done by simply configuring a C or Perl wrapper around script[3] for a standard sudo setup, (such as Owen's Perl script sudoscript [3]), I think that setup is less than optimal. I thought it would be nice if session information could be stored on a common, hardened server; additionally, I thought it would be nice not to have a C or Perl wrapper around script; finally, it would be nice to be able to customize other aspects of the process, such as the exact environmental variables passed, just as the wrapper script sudoscript does. It doesn't need to have the setuid bit set, since it is going to be invoked by sudo, so on its own it shouldn't be a security hazard; the recommended permission is to have it only executable (not readable or writable) by owner, and having root own it.

To effect this, I customized script to

- Write session transcripts to /var/log/super-trans, with each session in a separate file identified by the start time and the PID of the process.
- Write a keystroke log to syslogd (with the idea that syslog is configured to send these securely to another machine). The default setting currently is to use the facility LOCAL2, although there is a runtime option -F to let you (numerically) specify another facility.

3. script.c can be found in the RedHat source RPM util-linux-2.12a-16.EL4.6.src.rpm.

- Keep it fairly small and redistributable (it can be linked with dietlibc [7] to create a statically linked binary that is under 50k on a CentOS 4.2 distribution using gcc 3.4.4.)

To install suroot, all you need to do is compile suroot.c (available at http://www.cs.fsu.edu/~langley/suroot), place it in (for instance) /usr/local/bin owned by root and with permissions 0100 (execute bit only for root; it doesn't need to be suid), install one hard link per sudo user (for tracking purposes), and add a line to /etc/sudoers.

For instance, if after you install the binary in /usr/local/bin you want to let user1 use it, you would add this hard link:

```
ln /usr/local/bin/suroot /usr/local/bin/suroot-user1
```

and add the following line to /etc/sudoers:

```
user1 server1=/usr/local/bin/suroot-user1
```

The program suroot is simply a modification of script and keeps script's model. Here's how both script and suroot work, using three processes: (1) the original process, which is used for keyboard input (**parent_p**); (2) a child process, which is used for handling the output to the transcript (**child_p**); and (3) a grandchild (child of the child) process which is our shell (**gchild_p**).

Prior to creating **child_p** or **gchild_p**, we have **parent_p** clear all environmental variables except for TERM and HOME, and obtain a pseudo-terminal, either by the BSD standard openpty(3) or, if it isn't available, by searching for a free /dev/pty[p-s][0-9a-f] device.

Just before the **gchild_p** has a successful exec() to a shell process, its stdin, stdout, and stderr file descriptors are dup2()'ed over to the slave side of the pseudo-terminal. The current version of suroot uses a hard-coded /bin/bash as its shell; the shell is invoked with both the options -i (interactive) and -l (treat this as a login shell).

The **child_p** process has been modified slightly so that the transcript file is now always located in /var/log/super-trans/, and is named first by when the **child_p** process started and then by its PID. For example, the file name /var/log/super-trans/2006-01-20-18:06:38-021592 indicates that it was created on January 20, 2006, by process 21592.

To effect the system logging of keystrokes, the doinput() routine has been augmented with two new buffers, svbuf1 and svbuf2. The buffer svbuf1 records the raw input; if the process is in a default printable mode (non-printable characters are mapped into some visually attractive version, such as ASCII 010 being rendered as C-h), the printable contents of svbuf1 are copied into svbuf2. If raw characters are keystroke-logged, then, one would need to make sure that the receiving syslogd will be happy to receive them.

I like to statically link binaries such as this for three reasons:

- With a security application, I like to be certain that I am not using the wrong shared library; despite the care that sudo takes to make sure that all shared library paths are cleared from the environment (most importantly, of course, LD_LIBRARY_PATH), I am still leery of them.
- The functions that it is calling are simply not likely to be updated by any libc, so why bother to keep looking dynamically for updated versions of those functions every time that it runs?
- If you statically link with a small libc such as dietlibc, the resulting static binary is not much larger than the dynamic version.

However, static linking is not as easy these days as it might be in light of

the nss_* situation. I wanted to use glibc's getpwuid() to get home directory information; however, glibc's getpwuid() is now entangled with nss_*, which cannot be statically linked. I didn't want to write my own parser for the password file since, historically, this simple activity has been implicated in various security lapses, and I was already adding two new buffers that could potentially allow buffer overflows.

So I decided to go with a smaller, more compact libc that doesn't share this problem. For Linux I chose dietlibc, since I knew that it was complete enough to use for a full distribution (the Linux distribution DietLinux [8] is wholly built with dietlibc). I haven't managed yet to get suroot to statically link on Solaris. The implication here would be that somehow this would be started with UID 0 and a path such as LD_LIBRARY_PATH would somehow not be wiped out when the code removes all environmental variables except for TERM.

What are the limitations of this approach? The first is that this is only meant to directly run administrative code. It's not a general setup since it doesn't try to solve the problems of handling general users, such as those not in /etc/passwd or creating transcripts for non-root users (presently, transcripts are only created in /var/log/super-trans, which is only root writable). While both of these are addressable, there is a third problem (and one applicable also to the root account): the keystroke logging is not intelligent enough to detect the entry of a password, and will happily log any such that are typed.

REFERENCES

[1] Todd Miller, Chris Jepeway, Aaron Spangler, Jeff Nieusma, and Dave Hieb, Sudo Main Page, http://www.courtesan.com/sudo.

[2] Tom Christiansen and Dave Koblas, The op Wiki, https://svn.swapoff.org/op.

[3] Howard Owen, "The Problem of PORCMOLSULB," *;login:*, vol. 27, no. 4, August 2002.

[4] NSA, "Security Enhanced Linux," http://www.nsa.gov/selinux.

[5] Chet Ramey and Brian Fox, *GNU Bash Reference Manual* (Network Theory Ltd, 2003).

[6] The Honeynet Project, About the Project, http://www.honeynet.org.

[7] Felix von Leitner, "diet lib c—a libc optimized for small size," http://www.fefe.de/dietlibc/.

[8] Bernd Wachter, "Aardvarks DietLinux," http://www.dietlinux.org.