Executable based access control

Björn Bringert
bjorn@bringert.net
California Polytechnic State University

June 2003

Abstract

This paper presents an access control system where file access control can be based on the identity of the program that a process is running. This can be used to reduce the need for setuid root programs on UNIX systems. Instead of changing the user id of the process, the process is explicitly given access to the files it needs to access.

1 Introduction

In UNIX[1] operating systems, there is sometimes a need for unprivileged users to access files that the standard UNIX access control system does not allow them to access. One example of this situation is that users should be able to change their own passwords, but they cannot be given write permissions on the system password file. This problems is often solved by the use of setuid root programs, which run as root even if executed by an unprivileged user. If a setuid root program is vulnerable to, for example, buffer overflow attacks, an unprivileged user could use it to gain root privileges (commonly by having the program execute a shell, which will then run as root, Aleph One [1] has detailed information on how this may be done).

It should be clear that setuid root programs present a serious security problem and that to minimize the risk of security breaches, as few programs as possible should be setuid root. Some setuid root programs actually only need root privileges to access (typically to write to) a few files. Throughout this paper, we will use *passwd(1)* as an example of such a program. passwd only needs to be able to read and write the system password file, which is owned by root and only allows root to write to it.[2]

We propose a solution where programs running as unprivileged users can be granted extra permissions on individual files without being given full root privileges.

[1] In this paper, UNIX refers to UNIX™ and to UNIX-like operating systems such as Linux®

[2] This is somewhat simplified. In practice passwd may need to write to other files, such as temporary files and lock files. We also do not consider systems such as NIS [2], where passwords are not stored in a local file.

2 Model

We model our system as an *access matrix* [3, 4]. The subjects in our access matrix are executables and the objects are files. Thus, each cell of the access matrix corresponds to a triple

$\langle f, e, a\rangle : f \in files, e \in executables, a \in \wp(\{r, w, x\})$.

We call such a triple an *access cell* since it corresponds to a cell in the access matrix. The meaning of an access cell $\langle f, e, a\rangle$ is that any process running the executable e can be granted any set of permissions $a', a' \subseteq a$ on the file f.

Executables and files are represented by their absolute pathnames. An example of an access cell for passwd could be $\langle /etc/passwd, /usr/bin/passwd, \{r, w\}\rangle$. This would allow /usr/bin/passwd read and write access to /etc/passwd.

The system is complicated by the fact that one file may have multiple pathnames in UNIX. There can be several names (hard links) to a file and the same filesystem may be mounted in several places in the directory hierarchy. We define the function *aliases* such that:

$aliases(f) = \{f', f' \in pathnames : file(f) = file(f')\}$

In UNIX filesystems, file ownership and permissions are stored as attributes of a file, not of the directory entries that point to the file. It would be desirable that our system also have the property that the permissions on any alias of a file are identical. Furthermore, our system allows executables extra permissions in addition to those already given by the standard UNIX access control system. Thus, given an access matrix M, a process p is granted the permissions a on a file f if and only if at least one of the following conditions holds:

- p is given the permissions a on f by the standard UNIX access control system
- $\exists\langle f', e, a'\rangle \in M : f' \in aliases(f) \wedge e = executable(p) \wedge a \subseteq a'$

3 Implementation

The system was implemented by modifying a stock Linux 2.4.20 kernel [5]. In the Linux kernel, file access control is performed by the *permission()* function. This function uses the user id of the current process, the user and group ids of the file, and the permissions set of the file to decide whether the process should be granted the requested permissions on a file. The *permission()* function was modified to call *gran_suid_permission()*[3] if the normal permissions checking does not allow the process access to the file. Since this system can only grant more permissions than those given by the standard UNIX access controls, we need only consult the access matrix if the standard acess control system does not grant access. Thus our system adds virtually no overhead to the common case where access is granted by the normal UNIX access control system.

gran_suid_permission() takes a pointer to an inode and a permissions mask and returns 0 if access should be granted and -EACCES if it should be denied.

[3]The *gran_suid* part is an artifact from an earlier version of this system.

The function uses the directory entry cache and the list of mounted filesystems in the namespace of the process to obtain all aliases of the inode. The name of the executable is found by examining the virtual memory map of the process. For each file alias, the permissions for the file alias and the program named are looked up in the access matrix. Once a set of permissions has been found that allows the process the requested access to the file, access is granted. If no such permissions are found, access is denied.

gran_suid_permission() uses the directory entry cache to obtain the list of aliases of an inode. However, there is no guarantee that the alias for which the permissions have been set is in the diretory entry cache. This is a serious limitation, and it will cause the implementation to yield results that differ from those mandated by our model when the relevant alias in not in the directory entry cache. In practice this should not be very significant, since it is expected that the alias for which permissions have been set is the same one that is used to access the file. In the process of looking up the inode (which must be done before *permission()* can be called), the relevant directory entry must be read (and thus cached).

The access cells are stored in the kernel in an unbalanced binary search tree sorted by file and program path. Adding access cells from user space is done by writing to an entry in the /proc pseudo-filesystem (/proc/gran_suid). The current list of access cells can be obtained by reading from /proc/gran_suid. Only root is allowed to add access cells, but anyone can list them. The following format is used to represent access cells:

file:program:allow:permissions

where *file* is an absolute path to a file, *program* is the absoulte path to an executable, *allow* is the string "allow" (included to allow future expansions) and *permissions* is any ordered combination of the characters "r", "w" and "x", that is, one of the following strings:

$permissions \in \{$"rwx'', "rw'', "rx'', "r'', "wx'', "w'', "x'', "$''\}$

The example access cell used above would be written as:

"/etc/passwd:/usr/bin/passwd:allow:rw"

A new access cell specification overrides an existing one with the same file / program combination. Thus, the permissions granted by an existing access cell can be revoked by simply adding a new access cell using the same file / executable combination, but with empty permissions. In the current implementation the existing access cell is not removed when an access cell with the same file and executable but with empty permissions is added; instead it is simply overwritten.

Because of the simplistic way that entries are parsed, file and program names that contain the colon (:) character are currently not allowed.

A simple user space utility called *set_perms* has been provided to make adding permissions easier. It reads the file */etc/perms.conf*, removes blank lines and lines starting with "#", and writes the results to /proc/gran_suid.

The following is a somewhat contrived example session where unprivileged users running *cat(1)* are given read access to a file owned by root.

First we create a file /tmp/test owned by, and only readable by, root:

```
# echo hello > /tmp/test
# chmod 600 /tmp/test
# ls -l /tmp/test
-rw-------    1 root     root            6 Jun  4 11:44 /tmp/test
```

Then, as an ordinary user, we try to read it:

```
$ cat /tmp/test
cat: /tmp/test: Permission denied
```

Now, as root, we add an access cell that gives /bin/cat permission to read /tmp/test:

```
# echo "/tmp/test:/bin/cat:allow:r" > /proc/gran_suid
# cat /proc/gran_suid
/tmp/test:/bin/cat:allow:r
```

Now the ordinary user can read the file:

```
$ cat /tmp/test
hello
```

The above example is for demonstration purposes only. Since general purpose utilities like cat can read and write arbitrary information from and to files, the file owner could just as well have changed the normal permission bits on the files.

4 Testing

The system was tested by booting a Mandrake Linux 9.1 system, running on a uniprocessor x86 machine, with our patched 2.4.20 kernel. /usr/bin/passwd was copied to /tmp/passwd and the setuid bit on the executable was cleared. *strace(1)* was then used to determine which files passwd needs to access. The version of passwd that comes with Mandrake Linux 9.1 uses PAM (Pluggable Authentication Modules) [6] and the Shadow password suite [7]. In the process of changing a user's password, passwd was found to use the files /etc/.pwd.lock, /etc/shadow (the shadow password file), and /etc/nshadow (the new shadow file, passwd renames it to /etc/shadow). Since the files /etc/.pwd.lock and /etc/nshadow are created by passwd, passwd needs to have write and execute permissions on the /etc directory. Since passwd never actually writes to /etc/shadow (it simply unlinks it and links /etc/shadow to the newly created /etc/nshadow) write permissions on /etc/shadow are not needed. Thus it should be possible to run /tmp/passwd without it being setuid root if only $\langle /etc, /tmp/passwd, wx \rangle$ were added to the access matrix.

In order to test our hypothesis, we first set up a non-setuid version of passwd and add the access cell:

```
# cp /usr/bin/passwd /tmp
# ls -l /tmp/passwd
-r-x--x--x    1 root     root        16024 Jun  4 12:30 /tmp/passwd
# echo "/etc:/tmp/passwd:allow:wx" > /proc/gran_suid
# cat /proc/gran_suid
/etc:/tmp/passwd:allow:wx
```

Now, as an ordinary user, we try to change our password:

```
$ /tmp/passwd
Changing password for user bjorn.
Changing password for bjorn
(current) UNIX password:
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully.
```

The password was successful changed, and can be verified by logging in using the new password.

5 Advantages

Our system offers distinct advantages over using setuid root executables. If a flaw in a setuid root program can be exploited, the attacker may be able to execute arbitrary commands with full root privileges. With our system, the attacker would only be able to access the files to which the vulnerable program has been given access. Furthermore, if the attacker executes another executable (such as a shell), the process will run as the unprivileged user that ran the vulnerable program. Thus as standard buffer overflow attack such as that described by Aleph One [1] will fail to give the attacker a root shell. The executed shell would not even have the extra permissions granted to the vulnerable program, as the name of the binary will be different.

6 Drawbacks

If the files that the program is given access to affect system security, the given permissions could be used to gain root access to the system even if the program is not run setuid root. For example, a buffer overflow vulnerability in passwd could allow the attacker to execute code that would change the root password in the password file. This would be permitted by our system, as passwd has been given write access to the password file. The attacker could then simply log in as root using the new password.

Another drawback of the system is that in order to be able to create a file, the program must be given write permissions on the directory in which the file is to be created. Having write permissions on a directory allows the process to delete any file in that directory, something which is not neccessarily desirable.

If a program is executed setuid root because it needs access to some resource other than a file (for example if it needs to send a signal to a process not owned by the current user), our system cannot be used.

If an attacker is able to modify or replace an executable that has been given extra permissions, the new executable will be given those same permissions. This attack could be avoided if a system that prevents changes to executables were used. van Doorn [8] describes such a system for Linux.

Since we have decided to allow a program access to a file if the program has been given access to any alias of the file, an access matrix lookup must be performed for all aliases of the file in question. This is likely to degrade performance.

7 Related work

The Posix 1003.1e and 1003.2c Draft Standards [9, 10] propose a standard for extended access control. "The Extended attributes and access control lists" project [11] provides a Linux implementation based on these standards. The subjects in POSIX access control lists are users and groups, not executables.

The Linux trustees [12] project adds access control list support to linux. The access control is based on user and group id, not the identity of the executable that the process is running.

8 Alternative approaches

The normal UNIX access control can be used to provide similar results. For each file that requires extra privileges to access, we can create a separate group, give the group the required permissions and make each program that needs extra privileges on the file be setgid to that group. This approach does not work well when several programs need to access overlapping sets of files. Because a file can only be owned by one group and an executable only can be setgid to one group, all programs that access a given file must be in the same group and all files that a program accesses must be in the same group.

An access control list [13, 14] system can be used to grant a group access to a file and then make the program setgid to the group. This scales to multiple programs and files, but requires that an access control list system, such as POSIX ACLs [9, 10, 11] or Linux trustees [12] be deployed.

9 Future work

The performance impact of this system is potentially large, as an access matrix lookup may be performed every time the kernel needs to check whether a process has some set of permissions on a file. The need to check for permissions on all aliases of the file adds further overhead. The performance impact of this system thus needs to be studied.

In order to avoid the cost of repeatedly looking up the permissions in the access matrix, the permissions given to each executable on a file could be stored as attributes of the file. This would in effect be an access control list system with executables as subjects. The access control lists could be implemented using Linux extended attributes [11].

Another possible way of storing the access cells would be to store them as attributes of the executables, that is, each executable would have a list of files and permissions. This would essentially be a capability system. This approach suffers from the same file alias related problems as the system presented in this paper.

The parsing of access cell specifications needs to be modified to support filenames that contain colons and other special characters. This could be achieved by using escape sequences.

Currently the access cells are stored in an unbalanced binary search tree. This data structure was chosen for ease of implementation rather than performance. The unbalanced tree could be replaced by a hash table or balanced tree to improve its lookup performance.

When an empty access cell overrides an existing one, it would be desirable that the current cell be removed from the access matrix, and nothing be added.

10 Conclusions

Executable based access control seems to a be a viable approach to reducing the need for setuid root executables on UNIX systems. This approach is restricted to cases where the program only needs root privileges to access a predictable set of files.

It seems likely that a cleaner solution more in line with UNIX philosophy would be to store the permissions for each executable as attributes of the files that the executable needs to access. This should also yield better performance and move the implementation closer to our model.

References

[1] AlephOne, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, p. 14, November 1996. [Online]. Available: http://www.phrack.org/phrack/49/P49-14

[2] H. Stern, *Managing NFS and NIS*. 981 Chestnut Street, Newton, MA 02164, USA: O'Reilly & Associates, Inc., 1991.

[3] B. W. Lampson, "Protection," *ACM SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[4] P. J. Denning, "Third generation computer systems," *ACM Computing Surveys (CSUR)*, vol. 3, no. 4, pp. 175–216, 1971.

[5] L. Torvalds, "The linux kernel." [Online]. Available: http://www.kernel.org/

[6] V. Samar and R. Schemers, "Unified login with pluggable authentication modules. open software foundation request for comments 86.0," October 1995.

[7] J. F. Haugh, II, "Introduction to the shadow password suite," in *UNIX Security III Symposium, September 14–17, 1992. Baltimore, MD*, USENIX, Ed. Berkeley, CA, USA: USENIX, September 1992, pp. 133–144.

[8] L. van Doorn, G. Ballintijn, and W. A. Arbaugh, "Design and implementation of signed executables for linux," University of Maryland, Tech. Rep. CS-TR-4259, June 2001. [Online]. Available: http://www.cs.umd.edu/%7Ewaa/pubs/cs4259.ps

[9] C. Schaufler, Ed., *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)— Protection, Audit and Control Interfaces [C Language]*, ser. Draft Standard for Information technology—Portable Operating System Interface (POSIX). 345 E. 47th St, New York, NY 10017, USA: IEEE Computer Society, 1999, withdrawn draft.

[10] ——, *Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities— Protection and Control Interfaces*, ser. Draft Standard for Information technology—Portable Operating System Interface (POSIX). 345 E. 47th St, New York, NY 10017, USA: IEEE Computer Society, 1999, withdrawn draft.

[11] A. Grünbacher, "Extended attributes and acls for linux," 2003. [Online]. Available: http://acl.bestbits.at/

[12] V. Zavadsky, "Linux trustees," 2002. [Online]. Available: http://trustees.sourceforge.net/

[13] J. H. Saltzer, "Protection and control of information sharing in multics," in *Proceedings of the fourth symposium on Operating system principles*, 1973, p. 119.

[14] S. M. Kramer, "On incorporating access control lists into the unix operating system," in *UNIX Security Workshop Proceedings*. USENIX, 1988, pp. 38–48.