Race conditions, escalation attacks and SUID programs (with example)

Under most versions of UNIX, you can create shell scripts (or any interpreted scripts.) that are SUID or SGID. That is, you can create a shell script and, by setting the shell script's owner to be *root* and setting its SUID bit, you can force the shell script to execute with superuser privileges – thereby executing a privilege escalation attack.

This is because of a fundamental flaw with the UNIX implementation of shell scripts and – they give rise to a vulnerability caused by a *race condition*, a situation in which two processes execute simultaneously and wither one could finish first. Between the time that the Unix Kernel starts the script interpreter running, and the time that the script is opened for reading, it is possible for an attacker to replace the script that started the interpreter with another script.

SUID example: IFS and the /usr/lib/preserve hole

Sometimes, an interaction between a SUID program and a system program or library creates a security hole that's unknown to the author of the program. For this reason, it can be extremely difficult to know if a SUID program contains a security hole or not.

One of the most famous examples of a security hole of this type existed for years in the program called */usr/lib/preserve* (which is now given names similar to */usr/lib/ex3.5preserve*). This program, which is used by the vi and ex editors, automatically makes a backup of the file being edited if the user is unexpectedly disconnected from the system before writing out changes to the file. The preserve program writes the changes to a temporary file in a special directory, then uses the /bin/mail program to send the user a notification that the file has been saved.

Because people might be editing a file that was private or confidential, the directory used by the older version of the preserve program was not accessible by most users on the system. Therefore, to let the preserve program write into this directory, and let the recover program read from it, these programs were made SUID *root*.

Three details of the `/usr/lib/preserve` implementation worked together to allow knowledgeable system crackers to use the program to gain *root* privileges:

1. preserve was installed SUID root.
2. preserve ran */bin/mail* as the *root* user to alert users that their files had been preserved.
3. preserve executed the mail program with the system() function call.

The problem was that the system function uses *sh* to parse the string that it executes. There is a little-known shell variable called IFS, the internal field separator, which sh uses to figure out where the breaks are between words on

each line that it parses. Normally, IFS is set to the white space characters: space, tab, and newline. But by setting IFS to the slash character (/) then running vi, and then issuing the *preserve* command, it was possible to get */usr/lib/preserve* to execute a program in the current directory called bin. This program was executed as root. (/bin/mail got parsed as bin with the argument mail.)

If a user can convince the operating system to run a command as *root*, that user can become *root*. To see why this is so, imagine a simple shell script which might be called bin, and run through the hole described earlier:

```
#
# Shell script to make an SUID-root shell
#
cd /homes/mydir/bin
cp /bin/sh ./sh
# Now do the damage!
chown root sh
chmod 4755 sh
```

This shell script would get a copy of the Bourne shell program into the user's bin directory, and then make it SUID *root*. Indeed, this is the very way that the problem with */usr/lib/preserve* was exploited by system crackers.

The preserve program had more privilege than it needed - it violated a basic security principle called *least privilege*. Least privilege states that a program should have only the privileges it needs to perform the particular function it's supposed to perform, and no others. In this case, instead of being SUID *root*, */usr/lib/preserve* should have been SGID preserve, where preserve would have been a specially created group for this purpose. Although this restriction would not have completely eliminated the security hole, it would have made its presence considerably less dangerous. Breaking into the *preserve* group would have only let the attacker view files that had been preserved.

Although the preserve security hole was a part of UNIX since the addition of preserve to the vi editor, it wasn't widely known until 1986. For a variety of reasons, it wasn't fixed until a year after it was widely publicized.

NOTE: Another factor to consider is that, when using NFS, or any kind of remote filesystem mounting, it is not advisable to honour the suid bit if the program is on the remote filesystem. If honoured, this becomes an instant security loophole. This is why the use 0f the -nosuid flag is strongly advocated when using 'mount'.

Read more in
Garfinkel, “Practical Unix and Internet Security” (2003), pp. 149-150