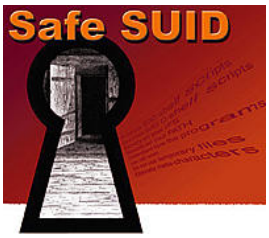
Dangers of SUID Shell Scripts



Thomas Akin

This article attempts to walk the fine line between full disclosure and published exploits. The object of this article is to illustrate how SUID programs work in order to help others writing their own programs avoid some common mistakes. The examples I provide are detailed enough to help you understand each danger, but I don't promise that all will work exactly as demonstrated if you try to use them maliciously. (sidebar)

Normally, UNIX scripts and programs run with the same permissions as the user who executes them. This is why typical users can't change their passwords by editing the /etc/passwd file; they don't have the permission to write to /etc/passwd, and no command they run will either. SUID programs, however, override normal permissions and always run with the permissions of the program's owner. Therefore, users can use the /usr/bin/passwd command to change their passwords. The /usr/bin/passwd command is SUID and is owned by root. It always runs with the same permissions as root.

When new administrators discover SUID, they often see it as a silver bullet that will solve all of their problems. They immediately begin using SUID scripts and programs to make their jobs easier. Unfortunately, they usually do it wrong.

When working with admins new to SUID, I often encounter scripts like this:

```
% ls change-pass
-rwsr-x---   1 root     helpdesk
    37 Feb 26 16:35 change-pass

% cat change-pass
#!/bin/csh -b
set user = $1
passwd $user
```

This simple script was set up to allow the help desk reset user passwords, which is a common need. The script is SUID root and is only executable by root or the members of the help desk group. This simple script is also riddled with holes. I'm going to expose seven of these holes and see whether they can be prevented. (sidebar)

The first problem occurs because this script is written in C-shell. C-shell scripts are vulnerable to manipulating environment variables. To take advantage of this, a hacker can compromise a help desk account (fairly trivial) and give himself a root shell with:

```
% env TERM='`cp /bin/sh /tmp/sh;chown root /tmp/sh;chmod 4755/tmp/sh`' change-pass
```

Lesson One -- Never use C-shell for SUID scripts.

```
% cat change-pass
#!/bin/ksh
user=$1
passwd $user
```

Rewriting the script in Korn shell helps us avoid the C-shell problem, but we still have problems. The script is vulnerable to a hacker manipulating the PATH variable. Because the program uses relative path names, a hacker can change his PATH to use his own program instead of the regular /usr/bin/passwd program:

```
% export PATH='/tmp'
% echo "cp /bin/sh /tmp/sh;chown root /tmp/sh;chmod 4755/tmp/sh" >/tmp/passwd
% ./change-pass
```

The PATH has been changed, and the change-pass command now runs the /tmp/passwd program instead of the /usr/bin/passwd program that we intended.

Lesson Two -- Always manually set the PATH and use absolute path names.

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
user=$1
/usr/bin/passwd $user
```

Now the PATH is secure and we are using absolute paths; but look closely and see that this script can change any password, even root's! We don't want the help desk (or a hacker) using our script to change root's password.

Lesson Three -- Understand how the programs in your script work.

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
user=$1
rm /tmp/.user
echo "$user" > /tmp/.user
isroot='/usr/bin/grep -c root /tmp/.user'
[ "$isroot" -gt 0 ] && echo "You Can't change root's password!" && exit
/usr/bin/passwd $user
```

Now this script will exit if someone enters root as the argument. But what happens if a hacker runs the program and doesn't specify an argument? The program will run the passwd command without any arguments. When the passwd command doesn't receive any arguments, it defaults to the current user. The problem is that in a root-owned SUID script, the current user is always root. The help desk (or hacker) can still change root's password by not giving change-pass any arguments.

Lesson Three (revised) -- Understand how the programs in your script work, especially how they handle arguments.

```
% cat change-pass
#!/bin/ksh
```

```
#!/bin/ksh
PATH='/bin:/usr/bin'
user=$1
[ -z $user ] && echo "Usage: change-pass username" && exit
rm /tmp/.user
echo "$user" > /tmp/.user
isroot='/usr/bin/grep -c root /tmp/.user'
[ "$isroot" -gt 0 ] && echo "You Can't change root's password!" && exit
/usr/bin/passwd $user
```

We no longer let anyone change root's password, but notice that we are using a temporary file. This script deletes the temporary file, recreates it, fills it with the username, and finally checks to see whether the username is root.

What if a hacker could time things perfectly so that just after the script removes the `/tmp/.user` file, but just before it creates a new `/tmp/.user` file, he created an empty `/tmp/.user` file? Would the hacker's file be overwritten? Possibly, but possibly not, depending on how file clobbering was set up. If the hacker's `/tmp/.user` is not overwritten, the hacker bypasses the checks and fools the script into changing root's password. To make this type of attack easier, a hacker could write a program that will automatically watch for activity and replace the `/tmp/.user` file.

Lesson Four -- Don't use temporary files! If you must use temporary files, don't put them in a publicly writable area.

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
user=$1
[ -z $user ] && echo "Usage: change-pass username" && exit
[ "$user" = root ] && echo "You can't change root's password!" && exit
/usr/bin/passwd $user
```

There are no temporary files, but now a hacker can use the well-known semi-colon trick. A semi-colon lets you put more than one command on a single line. By taking advantage of this, a hacker could type:

```
% change-pass "user;cp /bin/sh /tmp/sh;chown root /tmp/sh;chmod 4755 /tmp/sh"
```

Our script would take this input and run:

```
/usr/bin/passwd user;cp /bin/sh /tmp/sh;chown root /tmp/sh;chmod 4755 /tmp/sh
```

Each of these commands will be executed in order providing a root shell. To prevent problems like this, we need to make sure that any data the user inputs doesn't contain a semi-colon or any of the other shell meta-characters.

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
user=${1##*[ \\$/;()|\>\<&    ]}
[ -z $user ] && echo "Usage: change-pass username" && exit
[ "$user" = root ] && "You can't change root's password!" && exit
/usr/bin/passwd $user
```

We now remove any of the following characters from user input: a space, \, $, /, ;, (,), |, >, <, &, and a tab. It is difficult to see, but there is space after the open bracket ([ ) and a tab before the closing bracket ( ]).

Lesson Five -- Distrust and check all user input, and strip out any meta-characters.

Another common vulnerability is related to a shell's Internal Field Separator (IFS). The IFS specifies which characters separate commands. It is normally set to a space, tab, or new line. By changing the IFS, a hacker can change what programs our script executes. Our script calls the `/usr/bin/passwd` program. Changing the IFS to "/" with

```
% export IFS='/'
```

causes the script to no longer run `/usr/bin/passwd`, but instead run `usr bin passwd`. Now a hacker can create a script called `usr` that generates a root shell, and our SUID script will run that program for him.

Lesson Six -- Always manually set your IFS.

```
% cat change-pass
#!/bin/ksh
PATH='/bin:/usr/bin'
IFS=' '
user=${1##*[ \\$/;()|\>\<&    ]}
[ -z $user ] && echo "Usage: change-pass username" && exit
[ "$user" = root ] && "You can't change root's password!" && exit
/usr/bin/passwd $user
```

Unfortunately, we are still not safe. There is an inherent race condition in shell scripts that we can't fix with better programming. The problem is that running a shell script is a two-step process. First, the system starts up a new shell. Then, the new shell reads the contents of the script and executes it. By timing things perfectly, a hacker can exploit the delay between shell startup and when the script is read and executed. By creating a link to the SUID script:

```
% cd /tmp
% ln -s change-pass rootme
```

running the link, and quickly replacing it with another file:

```
% ./rootme &
% rm rootme && \
  echo "cp /bin/sh /tmp/sh;chown root /tmp/sh;chmod 4755 /tmp/sh" \

    >> rootme
```

it is possible to run anything as root. Done like this, there is only a small chance the attack would succeed, but there are techniques and programs to increase the chances of success and to automate the procedure for you. There are only two defenses against this attack. First, do not use SUID shell scripts. Second, some systems (e.g., Solaris) prevent this race condition by passing an open file handle to the new shell, thus avoiding the need to reopen and read the SUID file.

Lesson Seven -- Don't use SUID shell scripts.

Even after all our work, it is nearly impossible to create safe SUID shell scripts. (It is impossible on most systems.) Because of these problems, some systems (e.g., Linux) won't honor SUID on shell scripts. If you need the functionality of SUID, there are three more secure options. A wrapper program written in C, a Perl script, or a program like sudo. If you are new to secure programming, I recommend either sudo or a Perl script. SUID Perl scripts have built-in protection to prevent programmers from making the mistakes addressed in this article. For more information on secure SUID programming, see *Practical UNIX & Internet Security* (O'Reilly & Associates), or visit: http://seclab.cs.ucdavis.edu /~bishop/ \ secprog.html.

Thomas Akin has worked in Information Security for almost a decade and specializes in UNIX, Network, and Internet security. In addition to the CISSP, he has four UNIX and three Networking certifications. Most of his time is spent developing Information Security training programs, teaching, and writing. He can be reached at: takin@crossrealm.com.