4.7) How can I get setuid shell scripts to work?

[ This is a long answer, but it's a complicated and frequently-asked question. Thanks to Maarten Litmaath for this answer, and for the "indir" program mentioned below. ]

Let us first assume you are on a UNIX variant (e.g. 4.3BSD or SunOS) that knows about so-called `executable shell scripts'. Such a script must start with a line like:

```
#!/bin/sh
```

The script is called `executable' because just like a real (binary) executable it starts with a so-called `magic number' indicating the type of the executable. In our case this number is `#!' and the OS takes the rest of the first line as the interpreter for the script, possibly followed by 1 initial option like:

```
#!/bin/sed -f
```

Suppose this script is called `foo' and is found in /bin, then if you type:

```
foo arg1 arg2 arg3
```

the OS will rearrange things as though you had typed:

```
/bin/sed -f /bin/foo arg1 arg2 arg3
```

There is one difference though: if the setuid permission bit for `foo' is set, it will be honored in the first form of the command; if you really type the second form, the OS will honor the permission bits of /bin/sed, which is not setuid, of course.

---

OK, but what if my shell script does NOT start with such a `#!' line or my OS does not know about it?

Well, if the shell (or anybody else) tries to execute it, the OS will return an error indication, as the file does not start with a valid magic number. Upon receiving this indication the shell ASSUMES the file to be a shell script and gives it another try:

```
/bin/sh shell_script arguments
```

But we have already seen that a setuid bit on `shell_script' will NOT be honored in this case!

---

Right, but what about the security risks of setuid shell scripts?

Well, suppose the script is called `/etc/setuid_script', starting with:

```
#!/bin/sh
```

Now let us see what happens if we issue the following commands:

```
$ cd /tmp
$ ln /etc/setuid_script -i
$ PATH=.
$ -i
```

We know the last command will be rearranged to:

```
/bin/sh -i
```

But this command will give us an interactive shell, setuid to the owner of the script!
Fortunately this security hole can easily be closed by making the first line:

```
#!/bin/sh -
```

The `-' signals the end of the option list: the next argument `-i' will be taken as the name of the file to read commands from, just like it should!

---

There are more serious problems though:

```
$ cd /tmp
$ ln /etc/setuid_script temp
$ nice -20 temp &
$ mv my_script temp
```

The third command will be rearranged to:

```
nice -20 /bin/sh - temp
```

As this command runs so slowly, the fourth command might be able to replace the original `temp' with `my_script' BEFORE `temp' is opened by the shell!  There are 4 ways to fix this security hole:

1) let the OS start setuid scripts in a different, secure way - System V R4 and 4.4BSD use the /dev/fd driver to pass the interpreter a file descriptor for the script

2) let the script be interpreted indirectly, through a frontend that makes sure everything is all right before starting the real interpreter - if you use the `indir' program from comp.sources.unix the setuid script will look like this:

```
#!/bin/indir -u
#?/bin/sh /etc/setuid_script
```

3) make a `binary wrapper': a real executable that is setuid and whose only task is to execute the interpreter with the name of the script as an argument

4) make a general `setuid script server' that tries to locate the requested `service' in a database of valid scripts and upon success will start the right interpreter with the right arguments.

---

Now that we have made sure the right file gets interpreted, are there any risks left?

Certainly!  For shell scripts you must not forget to set the PATH variable to a safe path explicitly.  Can you figure out why? Also there is the IFS variable that might cause trouble if not set properly.  Other environment variables might turn out to compromise security as well, e.g. SHELL...  Furthermore you must make sure the commands in the script do not allow interactive shell escapes!  Then there is the umask which may have been set to something strange...

Etcetera.  You should realise that a setuid script `inherits' all the bugs and security risks of the commands that it calls!

All in all we get the impression setuid shell scripts are quite a risky business!  You may be better off writing a C program instead!