When is a command line not a line?

Tim Waugh

<twaugh@redhat.com>

Revision History

Revision 1	19 October 2001	TMW
Initial revision.		
Revision 2	6 November 2001	TMW
Mention that Perl's system() can take a list.		

Abstract

This article discusses the way in which intuitive handling of 'command lines' can lead to bugs and security problems, and suggests a solution.

An intuitive idea

We all know what a command line is. It's what you give to the shell in order to run a program with, perhaps, some options. It's what you type in to make something happen. When you are typing in a command line, you do it character by character, separating the command's name and its options by spaces. And that's all there is to it.

At a programming level, it's the same thing: Perl has `system()`; so does C; Python has `os.system()`; all of which take a command line (a string of characters) and execute it.

So what's the point of writing this article? The answer is that all is not as it seems. When it comes to actually finding the command to execute, the 'command line' must be broken up into an array of *words*. Here's why:

```
int execve(const char *filename,
           char *const argv[],
           char *const envp[]);
```

The `execve()` C library function that eventually gets called takes an *array* (or *vector*) of strings, *`argv`*, not a single command line string. This array of strings is given to the program when it starts up, and it appears as one of the parameters to the `main()` entry point function. We'll refer to this as the *argument vector*, as distinct from the *command line* which is the string of characters we started with.

So far we've seen two different ways of looking at command lines: a string of characters, and an array of strings. You might be thinking that really they are the same and that you can get from one to the other without any problems. You would be largely right, but it isn't as easy to get from one to the other as you might think. That's where the problems start to appear.

String replacements in command lines

Often a program will want to change a parameter in a command line; an example would be to run a command on a user-supplied file. To make it more concrete, let's say that our program converts DVI files

to PostScript and to PDF. It does so by enlisting the help of two other programs, **dvips** and **dvipdf**. These commands are to be invoked as follows:
`dvips -o output input`
`dvipdf input output`

If we want to write this program using a POSIX shell-like language, it would probably look something like Figure 1.

Figure 1. Implementation using Bash

```
#!/bin/bash
# $1 is the file we want to convert
dvips -o out.ps "$1"
dvipdf "$1" out.pdf
```

`$1` is a *positional parameter*: when the program is run with arguments, it takes on the value of the first argument—in fact, the first word, other than the command itself.

When this program is run with the command **`convert "my file.dvi"`**, `$1` will be replaced with `my file.dvi`, and so the command lines executed will be:

- `dvips -o out.ps "my file.dvi"`
- `dvipdf "my file.dvi" out.pdf`

In terms of argument vectors, these translate to:

- [‘`dvips`’, ‘`-o`’, ‘`out.ps`’, ‘`my file.dvi`’]
- [‘`dvipdf`’, ‘`my file.dvi`’, ‘`out.pdf`’]

So long as we don’t care about checking for errors, this implementation is exactly what we wanted. It will even cope with spaces in the file name, because the double quotes will keep it all together as one ‘word’, hence one entry in the array of strings given to `main()`. (Of course, in order to be useful both **dvips** and **dvipdf** would need to handle spaces in filenames too.) Easy enough, so now let’s try that in Python using `os.system()`.

Figure 2. Implementation using Python

```
#!/usr/bin/python
import os
import sys
os.system ('dvips -o out.ps "' + sys.argv[1] + '"')
os.system ('dvipdf "' + sys.argv[1] + '" out.pdf')
```

(Here, `sys.argv[1]` is the same string as `$1` was before.)

Although it might look the same, this implementation in Python is quite different to the one in Figure 1. Imagine if the file name happens to contain a dollar sign (by chance or by malice). If by malice, the miscreant can use parentheses in the (made up) file name to execute commands of their own—command substitution of '`$(command)`' will cause the command to be executed. With the Bash implementation, this was not possible because the malicious file name was in an environment variable and substituted into the command line using parameter expansion—command substitution is always performed before this step. Obviously with this small example there isn't a lot of scope for malice, but there are plenty of examples where there is: web-based interfaces, mail and print filters, and so on.

To try to fix this up, we could try to add our own quoting to the argument: whenever we find a dollar sign, replace it with a backslash followed by a dollar sign (this takes away its special meaning). Are we safe yet? Well, no, because now the miscreant can still use backticks (backticks are another form of command substitution), or just break out of the quotation by using their own quotation marks in the file name such as '`"; cat /etc/passwd; echo "`', and still get their own programs run.

So we need to, for want of a real word, enquote the file name, escape any backticks we find, and escape any dollar signs we find. Or more efficiently, we could use single-quote characters instead of double-quote characters, and escape any stray single-quote characters in the file name (command substitutions and parameter expansions are not performed inside single-quoted words).

An easier way of doing it is presented in Figure 3.

Figure 3. Alternative implementation in Python

```
#!/usr/bin/python
import os
import sys
os.environ['file']=sys.argv[1]
os.system ('dvips -o out.ps "$file"')
os.system ('dvipdf "$file" out.pdf')
```

With this implementation, the shell (which is what `os.system` invokes in this instance) does the tricky stuff for us.

Where we went wrong before was in trying to modify the command line that we had. When the shell interprets the command line, it needs to apply quoting rules to it, and parameter expansion rules, and command substitution rules, and several others, before it is finally split into the words that make up the *argument vector*, the *`argv`* parameter that gets given to `main()`. Trying to insert arbitrary strings into a command line without knowing these rules can end up causing problems, as we've seen.

Attempts to fix the problem

The `system()` function, then, should always be given a fixed command line, with any unknowns filled in by the shell using parameter expansion, to be sure of getting the correct results. Even better is to use `execve()` directly, if possible. (Perl provides a way of doing this: `system()` can take a list.)

This isn't always possible, of course: in some cases, the command line to use will be configurable, with the syntax for indicating replaceable parameters already fixed, as we'll see in the section called "Desktop Entries". But even in those cases, it is possible to transform the problem into one that can be solved using a known command line (or at least, one constructed from known components).

If you are going to be writing some code that you think needs to modify a command line, there are some library functions around that seem like they could help you avoid making mistakes in the implementation. Let's take a look at some of the candidates.

In the popt package, there is a function named `poptParseArgvString()`, which will convert a command line into an argument vector 'using rules similar to normal shell parsing'. Unfortunately, parameter expansion is not supported, and so it is no use the for job at hand.

Similarly, glib has a function named `g_shell_parse_argv()`, but it also misses the point: parameter expansion is not supported.

To find a function that really does the job, we need look no further than POSIX, which specifies the `wordexp()` function. (For a detailed discussion of the `wordexp()` function, see the GNU C library manual at <http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_150.html#SEC159>). This function *does* perform parameter expansion, and a lot more besides. It can also be told not to perform command subsitution, for additional safety.

Real examples of command line manipulation

Mailcap

RFC 1524 at <http://www.faqs.org/rfcs/rfc1524.html> defines a file format for mail user agents to use in order to work out how to handle different MIME types. A typical entry in a mailcap file might look like this:

```
application/pgp; gpg < %s | metamail; needsterminal; \
       test=test %{encapsulation}=entity ; copiousoutput
application/pgp; gpg < %s; needsterminal ; copiousoutput
```

Here we can see a replaceable string: `%s`. You might think that the specification would be robust against malicious mail received by an innocent mail user agent, and would offer the implementor advice on how to avoid accidentally running a command embedded inside the mail as a result of trying to execute a mailcap command. Here's what it says:

> Security issues are not discussed in this memo.
>
> — RFC 1524, Security Considerations

To be fair, it was written in 1993, when the Internet was a friendlier place than it is now. Nevertheless, one would think it reasonable to assume that some discussion of how to interpret the mailcap entries would involve a step-by-step procedure for the string replacements. Here are the relevant pieces:

> (Because of differences in shells and the implementation and behavior of the same shell from one system to another, it is specified that the command line be intended as input to the Bourne shell, i.e., that it is implicitly preceded by `"/bin/sh -c "` on the command line.)
>
> The two characters `"%s"`, if used, will be replaced by the name of a file for the actual mail body data. [...] Furthermore, any occurrence of `"%t"` will be replaced by the content-type and subtype specification. (That is, if the content-type is `"text/plain"`, then `%t` will be replaced by `"text/plain"`.) A literal % character may be quoted as \%. Finally, named parameters from the Content-type field may be placed in the command execution line using `"%{"` followed by the parameter name and a closing `"}"` character. The entire parameter should appear as a single command line argument, regardless of embedded spaces.
>
> — RFC 1524, Appendix A

The author has tried to be quite specific about what to do, but (unfortunately) not to the point of being pedantic about it. There are several problems with this text:

- The part about prefixing the entire command line with `"/bin/sh -c "` was obviously an attempt to clarify how the command line is to be interpreted, but really I think it has made things more confusing. Is the entire mailcap-specified command line to be quoted or left as is before prefixing, and if quoted, then how: single quotes, or double quotes? The intent, I think, was to say "Bourne shell quoting and field-splitting rules apply" but already the damage is done: the implementor, reading the RFC, is now thinking that they need to manipulate the command line somehow. So, with that seed of confusion already in their mind...
- ...they are told to replace occurrences of `%s` with things. The quoting is quite confused in appendix A: `%s` is shown quoted; later `%t` is shown with no quotes; and `%t`'s example replacement text, is then quoted. If it's intentional, it's pretty subtle, but noticeable enough to add to the confusion.
- Good news near the end though, as `%{...}` is to be replaced with "a single command line argument", implying one word in an argument vector. But what about our example above, "`test %{encapsulation}=entity`"? Is this to be replaced with ["`test`", "`encaps-value`", "`=entity`"]? It may be what's intended, but it isn't clear and it may matter.
- There is no such talk of "single command line arguments" for `%s` or `%t` substitutions, and so we are left guessing about similar questions of those.
- How are shell pipeline and sequence characters to be interpreted? Need they be quoted? Are they disallowed because they are not mentioned? (If so, our gpg example above will not work.)

In short, it isn't clear exactly how to construct an argument vector from a mailcap command line. Fortunately, the mail user agent gets to choose the temporary file name to use in place of `%s`, and in general mail user agents seem to replace awkward characters (such as quotes, dollar signs, etc) in other possible replacement strings with harmless characters (such as underscores) instead.

Nevertheless, some sample `mailcap` files have entries like this:

```
text/plain; shownonascii iso-8859-8 %s | more ; \
  test=test "‘echo %{charset} | tr ’[A-Z]’ ’[a-z]’‘" = iso-8859-8; \
```

```
needsterminal
```

This is exploitable. The problem with this is that `%{charset}` is not quoted, possibly because the author was unsure of how mail user agents would deal with string replacement in that case—and all because the RFC wasn't clear enough on this point.

It would probably have been a good idea if the specification had instead said something along these lines: the environment variable `s` is given the value of the name of the temporary file name containing the data to process; `t` is given the value of the MIME type of the data; and `field_`*`fieldname`* (for each *`fieldname`* in the Content-type header) are each given the value of the named Content-type field. The command line *`cmd`* is then executed using the argument vector ["`/bin/sh`", "`-c`", *`cmd`*].

Desktop Entries

Desktop environments such as KDE and GNOME use short configuration files known as *desktop entries*, which describe how an application is to be launched, what it's name is for menus, and so on. The Desktop Entry Standard at <http://www.freedesktop.org/standards/desktop-entry-spec/desktop-entry-spec.html> is currently being drafted. One of the fields in the configuration file is `Exec=`, and it describes the command line to use to launch the program.

It has *parameter variables*, much like shell environment variables. For example, `%f` is replaced by a single file name if a file is dragged onto an icon representing a desktop entry, and `%F` is replaced by a list of files in case a selection of files is dropped onto the icon at once. What is unfortunate is that the manner in which these parameter variables are to be replaced is not specified—at least one implementation has transformed the command line "`appname %f`" into the argument vector ["`appname`", ""] when there is no file name to substitute—and there is no discussion of quoting issues at all.

It's all very similar to the mailcap case, and since different desktop environment implementations (such as GNOME and KDE) need to understand how to convert a command line into an argument vector (and application packagers need to understand how to write desktop entry files in the first place), it is important to have a specification that is clear on these issues. It isn't enough to have all implementations agree: people have to write these files and so it needs to be documented.

The desktop entry file has a problematic requirement though. Not all parameter variables can be replaced by environment variables and given to the shell to deal with. Some parameter variables are lists.

POSIX shell parameter expansions have a nasty flaw in that they don't really allow for lists like this, except in one special case. The issue here is spaces, or more accurately "inter-field separators" (spaces, tabs, and newlines), which are the characters that are used to separate words in a command line. We've seen how putting double-quotes around a parameter expansion makes sure that the end result is contained in one word in the argument vector: `F="my file.dvi"; ls $F` won't show me my DVI file, but `F="my file.dvi"; ls "$F"` will. Positional parameters are slightly special because they are numbered, and we don't know in advance how many of them there are. The special string `$*` undergoes the following parameter expansion: it is replaced by *all* of the positional parameters, i.e. all of the arguments. Unfortunately field splitting happens afterwards and so any values containing spaces will get broken up into multiple words. But quoting doesn't help: `"$*"` is a single word containing all of the arguments, which isn't very useful. So `$@`, another special string, acts the same as `$*` except when enclosed in double-quotes, in which case it undergoes parameter expansion to produce one word per positional parameter. So

if you want to pass the arguments on to another program, `other_program "$@"` is a good way to do it.

That's the only way to do it, and all you get are the positional parameters. With desktop entry files, there are several different list-type parameter variables, so they can't all use that mechanism.

One way around that is to use `wordexp()` first to do the hard parts of word expansion, and then manipulate the argument vector afterwards. Manipulating argument vectors is less error prone than manipulating command lines because there is no special interpretation to be done afterwards. Just call `execve()`, and your argument vector lands up in the executed program's `main()` function.

A good design for desktop entry `Exec=` lines would probably involve the following steps for transforming the text after the equal sign into an argument vector:

First, a couple of restrictions on the allowable values. All of the list variables (such as `%F`) must be on their own where they appear; they must not appear next to any character except a space or either end of the line. The string "`/../`" is not permitted in the command line string.

1. For each variable occurring in the string that is not a list variable, if the percent sign is not immediately prefixed by an odd number of backslash characters, replace both the percent sign and its variable letter *x* with `${x}`. For example, replace `%f` with `${f}`.

2. For each variable occurring in the string that is a list variable, if the percent sign is not escaped as in the previous step, replace both the percent sign and its variable letter *X* with `/../X`.

3. Call `wordexp()` to split the command line into words and form an argument vector, taking care to pass the flag that instructs `wordexp()` to return an error if command substitution would be performed.

4. Build a new argument vector by copying, word by word, the one created by the call to `wordexp()`. For each word that starts "`/../`", replace it with one word for each of the values that the relevant variable is a list of.

The string "`/../`" was chosen for marking the list variables because it is not subject to expansion or substitution, and while formed of valid file name characters, is not a useful file name itself. My suggestion for handling desktop entries does not include executing the shell to handle pipelines and sequences of commands as in the section called "Mailcap"; there seems little need in this application.

Conclusion

It seems clear (to me at least) that thinking in terms of a command line, a string of characters, can lead to problems and that thinking in terms of an *argument vector*, a list of character strings (the *`argv`* parameter to `main()`) is a robust way to avoid those problems. At the very least, if a command line must be involved, it should not be tampered with unaided: leave it to the shell, or to `wordexp()`, or at the very least make sure that you are replacing known strings with known strings.

The moral of the story, then, is that the POSIX committee spent a lot of time thinking about these types of issues when they drafted the specification of `wordexp()`, and so we should learn from their wisdom.