Fixing Unix/Linux/POSIX Filenames: Control Characters (such as Newline), Leading Dashes, and Other Problems

David A. Wheeler

2014-12-19 (originally 2009-03-24)

Seek freedom and become captive of your desires, seek discipline and find your liberty. — Frank Herbert, Dune

"Negative freedom is freedom from constraint, that is, permission to do things; Positive freedom is empowerment, that is, ability to do things... Negative and positive freedoms, it might seem, are two different descriptions of the same thing. No! Life is not so simple. There is reason to think that constraints (prohibitions, if you like) can actually help people to do things better. Constraints can enhance ability..." — Angus Sibley, "Two Kinds of Freedom"

*"...filesystem people should aim to make "badly written" code "just work" unless people are really really unlucky. Because like it or not, that's what 99% of all code is... Crying that it's an application bug is like crying over the speed of light: you should deal with *reality*, not what you wish reality was." — Linus Torvalds, on a slightly different topic (but I like the sentiment)*

Traditionally, Unix/Linux/POSIX pathnames and filenames can be almost any sequence of bytes. A *pathname* lets you select a particular file, and may include zero or more "/" characters. Each pathname component (separated by "/") is a filename; filenames cannot contain "/". Neither filenames nor pathnames can contain the ASCII NUL character (\0), because that is the terminator.

This lack of limitations is flexible, but it also creates a legion of unnecessary problems. In particular, this lack of limitations makes it unnecessarily difficult to write correct programs (enabling many security flaws). It also makes it impossible to consistently and accurately display filenames, causes portability problems, and confuses users.

This article will try to convince you that adding *some* tiny limitations on legal Unix/Linux/POSIX filenames would be an improvement. Many programs *already* presume these limitations, the POSIX standard *already* permits such limitations, and many Unix/Linux filesystems *already* embed such limitations — so it'd be better to make these (reasonable) assumptions true in the first place. This article will discuss, in particular, the three biggest problems: control characters in filenames (including newline, tab, and escape), leading dashes in filenames, and the lack of a standard character encoding scheme (instead of using UTF-8). These three problems impact programs written in any language on Unix/Linux/POSIX system. There are other problems, of course. Spaces in filenames can cause problems; it's probably hopeless to ban them outright, but resolving some of the other issues will simplify handling spaces in filenames. For example, when using a Bourne shell, you can use an IFS trick (using `IFS=`printf '\n\t'``) to eliminate some problems with spaces. Similarly, special metacharacters in filenames cause some problems; I suspect few if any metacharacters could be forbidden on all POSIX systems, but it'd be great if administrators could locally configure systems so that they could prevent or escape such filenames when they want to. I then discuss some other tricks that can help.

After limiting filenames slightly, creating completely-correct programs is **much** easier, and some vulnerabilities in existing programs disappear. This article then notes some others' opinions; I knew that some people wouldn't agree with me, but I'm heartened that many *do* agree that something should be done. Finally, I briefly discuss some methods for solving this long-term; these include forbidding creation of such names (hiding them if they already exist on the underlying filesystem), implementing escaping mechanisms, or changing how tools work so that these are no longer problems (e.g., when globbing/scanning, have the libraries prefix "./" to any filename beginning with "-"). Solving this is not easy, and I suspect that several solutions will be needed. In fact, this paper became long over time because I kept finding new problems that needed explaining (new "worms under the rocks"). If I've convinced you that this needs improving, I'd like your help in figuring out how to best do it!

Filename problems affect programs written in any programming language. However, they can be especially tricky to deal with when using Bourne shells (including bash and dash). If you just want to write shell programs that can handle filenames correctly, you should see the short companion article Filenames and Pathnames in Shell: How to do it correctly.

1. A tale of complications

Imagine that you don't know Unix/Linux/POSIX (I presume you really do), and that you're trying to do some simple tasks. For our purposes we will create simple scripts on the command line (using a Bourne shell) for these tasks, though many of the underlying problems affect *any* program. For example, let's try to print out the contents of all files in the current directory, putting the contents into a file in the parent directory:

```
cat * > ../collection  # WRONG
```

In a well-designed system, simple things should be simple, and the "obvious easy" way to do simple common tasks should be the correct way. I call this goal "*no sharp edges*" — to use an analogy, if you're designing a wrench, don't put razor blades on the handles. Typical Unix/Linux filesystems fail this test — they *do* have sharp edges.

The list doesn't include "hidden" files (filenames beginning with "."), but often that's what you want anyway, so that's not unreasonable. The problem with this approach is that although this *usually* works, filenames could begin with "-" (e.g., "-n"). So if there's a file named "-n", and you're using GNU cat, all of a sudden your output will be numbered! Oops; that means on *every* command we have to disable option processing. The "obvious" way to do this is to litter command invocations with "--" before the filename(s). But it turns out this doesn't really work, because not all commands support "--" (ugh!). For example, the widely-used "echo" command is not required to support "--". What's worse, echo does support at least one dash option, so we need to escape leading-dash values somehow. POSIX recommends that you use printf(1) instead of echo(1), but some old systems do not include printf(1). In my opinion, a much better solution is to prefix globs like this with "./". In other words, you should do this instead:

```
cat ./* > ../collection  # CORRECT
```

Prefixing relative globs with "./" always solves the "leading dash" problem, but it sure isn't obvious. In fact, many shell books and guides completely omit this information, or don't explain it until far later in the book (which many people never read). Even people who know this will occasionally forget to do it. After all, people tend to do things the "easy way" that *seems* to work, resulting in millions of programs that have subtle bugs (which sometimes lead to exploits). Complaining that people must rewrite all of their programs to use a non-obvious (and ugly) construct is unrealistic. Most people who write `cat *` do not intend for the filenames to be used as command options (as noted in the *The Unix-haters Handbook* page 27).

In many cases globbing isn't what we want. We probably don't want the "cat *" command to examine directories, and glob patterns like "*" won't recursively descend into subdirectories either. The primary tool for walking POSIX filesystems is the "find" command. In theory, we could just replace the "*" with something that computes the list of such file names (which will also include the hidden files):

```
cat `find . -type f` > ../collection  # WRONG
```

This construct doesn't fail because of leading dashes; find always prefixes filenames with the starting directory, so all of the filenames in this example will start with "./". This construct does have trouble with scale — if the list is really long, you risk an "argument list too long" error, and even if it works, the system has to build up a complete list all at once (which is slow and resource-consuming if the list is long). Even if the list of files is short, this construct has many other problems. One problem (among several!) is that if filenames can contain spaces, their names will be split (file "a b" will be incorrectly parsed as two files, "a" and "b").

Okay, so let's use a "for" loop, which is better at scaling up to large sets of files and complicated processing of the results. When using shell you need to use `set -f` to deal with filenames containing glob characters (like asterisk), but you can do that. Problem is, the "obvious" for loop won't work either, for the same reason; it breaks up filenames that contain spaces, newlines or tabs:

```
( set -f ; for file in `find . -type f` ; do  # WRONG
    cat "$file"
  done ) > ../collection
```

How about using find with a "while read" loop? Let's try this:

```
( find . -type f |    # WRONG
  while read filename ; do cat "$filename" ; done ) > ../collection
```

This is widely used, but still wrong. It works if a filename has spaces in the middle, but it won't work correctly if the filename begins or ends with whitespace (they will get chopped off). Also, if a filename includes "\", it'll get corrupted; in particular, if it ends in "\", it will be combined with the next filename (trashing both). Okay, maybe that's just a perversity of the defaults of shell's "read", but there are other problems as we'll see in a moment.

Now at this point, some of you may suggest using xargs, like this:

```
( find . -type f | xargs cat ) > ../collection # WRONG, WAY WRONG
```

Yet this is wrong on many levels. If you try to use xargs, and limit yourself to the POSIX standard, xargs is *painful* to use. By default, xargs' input is *parsed*, so space characters (as well as newlines) separate arguments, and the backslash, apostrophe, double-quote, *and* ampersand characters are used for quoting. According to the POSIX standard, underscore *may* have a special meaning (it will stop processing) if you omit the -E option, too! So even though this "simple" use of xargs works on some filenames, it fails on many characters that are allowed in filenames. The xargs quoting convention isn't even consistent with the shell. Using xargs while limiting yourself to the POSIX standard is an exercise in pain, if you are trying to create actually-correct programs, because it requires substitutions to work around xargs quoting.

So let's "fix" handling filenames with spaces by combining find (which can output filenames a line at a time) with a "while" loop (using read -r and IFS), a "for" loop, xargs with quoting and -E, or xargs using a non-standard GNU extension "-d" (the extension makes xargs more useful):

```
# WRONG:
 ( find . -type f |
   while IFS="" read -r filename ; do cat "$filename" ; done ) > ../collection

 # OR WRONG:
  IFS="`printf '\n'`"   # Split filenames only on newline, not space or tab
  ( for filename in `find . -type f` ; do
      cat "$filename"
    done ) > ../collection

 # OR WRONG, yet portable; space/backslash/apostrophe/quotes ok in filenames:
 ( find . -type f | sed -e 's/[^[:alnum:]]/\\&/g' |
   xargs -E "" cat ) > ../collection

 # OR WRONG _and_ NON-STANDARD (uses a GNU extension):
 ( find . -type f | xargs -d "\n" cat ) > ../collection
```

Whups, all four of these don’t work correctly either. All of these create a list of filenames, with each filename terminated by a newline (just like the previous version of “while”). But **filenames can include newlines!**

Handling filenames with all possible characters (including newlines) is often hard to do portably. You can use `find...-exec...{}`, which is portable, but this gets ugly fast if the command being executed is nontrivial. It can also be slow, because this has to start a new process for every file, and the new process cannot trivially set a variable that can be used afterwards (the variable value disappears when the process goes away). POSIX has more recently extended find so that find -exec ... {} + (plus-at-end) creates sets of filenames that are passed to other programs (similar to how xargs works); this is faster, but it still creates new processes, making tracking-while-processing very inconvenient. I believe that some versions of find have not yet implemented this more recent addition, which is another negative to using it (but it *is* standard so I expect that problem to go away over time). In any case, both of these forms get ugly fast if what you’re exec-ing is nontrivial:

```
  # These are CORRECT but have many downsides:
  ( find . -type f -exec cat {} \; ) > ../collection
  # OR
  ( find . -type f -exec cat {} +  ) > ../collection
```

If you use GNU find and GNU xargs, you can use non-standard extensions to separate filenames with \0 instead:

```
 # CORRECT but nonstandard:
 ( find . -type f -print0 | xargs -0 cat ) > ../collection
 # OR, also correct but nonstandard:
 find . -print0 |
 while IFS="" read -r -d "" file ; do ...
   # Use "$file" not $file everywhere.
 done
```

But using \0 as a filename separator requires that you use non-standard (non-portable) extensions, this convention is supported by only a few tools, and the option names to use this convention (when available) are jarringly inconsistent (perl has -0, while GNU tools have sort -z, find -print0, xargs -0, and grep either -Z or --null). This format is also difficult to view and modify (in part because so few tools support it), compared to the line-at-a-time format that is widely supported. You can’t even pass such null-separated lists back to the shell via command substitution; `` cat `find . -print0` `` and similar “for” loops don’t work. Even the POSIX standard’s version of “read” can’t use \0 as the separator (POSIX’s read has the -r option, but not bash’s -d option), so they’re really awkward to use. This is silly; processing lines of text files is well-supported, and filenames are an extremely common data value, but you can’t easily combine these constructs?

Oh, and ***don’t display filenames***. Filenames could contain control characters that control the terminal (and X-windows), causing nasty side-effects on display. Displaying filenames can even cause a security vulnerability — and who expects *printing a filename* to be a vulnerability?!? In addition, you have no way of knowing for certain what the filename's character encoding is, so if you got a filename from someone else who uses non-ASCII characters, you're likely to end up with garbage mojibake.

Ugh — lots of annoying problems, caused not because we don't have enough flexibility, but because we have too much. Many documents describe the complicated mechanisms that can be used to deal with this problem, such as BashFAQ's discussion on handling newlines in filenames. Many of the suggestions posted on the web are *wrong*, for example, many people recommend the incorrect `while read line` as the correct solution. In fact, I found that the BashFAQ's 2009-03-29 entry didn't walk files correctly either (one of their examples used `for file in *.mp3; do mv "$file" ...`, but this fails if a filename begins with "-"; yes, I fixed it). If the "obvious" approaches to common tasks don't work correctly, and require complicated mechanisms instead, I think there is a *problem*.

In a well-designed system, simple things should be simple, and the "obvious easy" way to do simple common tasks should be the correct way. I call this goal "*no sharp edges*" — to use an analogy, if you're designing a wrench, don't put razor blades on the handles. Typical Unix/Linux filesystems fail this test — they *do* have sharp edges. Because it's hard to do things the "right" way, many Unix/Linux programs simply assume that "filenames are reasonable", even though the system doesn't guarantee that this is true. This leads to programs with occasional errors that are sometimes hard to solve.

In some cases, these errors can even be security vulnerabilities. My "Secure Programming for Linux and Unix HOWTO" has a section dedicated to vulnerabilities caused by filenames. Similarly, CERT's "Secure Coding" item MSC09-C (Character Encoding — Use Subset of ASCII for Safety) specifically discusses the vulnerabilities due to filenames. The Common Weakness Enumeration (CWE) includes 3 weaknesses related to this (CWE 78, CWE 73, and CWE 116), all of which are in the 2009 CWE/SANS Top 25 Most Dangerous Programming Errors. Vulnerability CVE-2011-1155 (logrotate) and CVE-2013-7085 (uscan in devscripts, which allowed remote attackers delete arbitrary files via a whitespace character in a filename) are a few examples of the many vulnerabilities that can be triggered by malicious filenames.

These types of vulnerabilities occasionally get rediscovered, too. For example, Leon Juranic released in 2014 an essay titled Back to the Future: Unix Wildcards Gone Wild, which demonstrates some of the problems that can be caused because filenames can begin with a hyphen (which are then expanded by wildcards). I am really glad that Juranic is making more people aware of the problem! However, this is not new information; these types of vulnerabilities have been known for decades. Kucan comments on this, noting that this particular vulnerability can be countered by always beginning wildcards with "./". This is true, and for many years I have been recommended prefixing globs with "./". I still recommend it as part of a solution that works today. However, we've been trying to teach people to do this for decades, and the teaching is not working. People do things the easy way, even if it creates vulnerabilities.

It would be better if the system actually *did* guarantee that filenames were reasonable; then already-written programs would be correct. For example, if you could guarantee that filenames don't include control characters and don't start with "-", the following script patterns would always work correctly:

```
#!/bin/sh
# CORRECT if files can't contain control chars and can't start with "-":
set -eu                # Always put this in Bourne shell scripts
IFS="`printf '\n\t'`"  # Always put this in Bourne shell scripts
```

```
# This presumes filenames can't include control characters:
for file in `find .` ; do ... command "$file" ...  done
# This presumes filenames can't begin with "-":
for file in *        ; do ... command "$file" ...  done
# You can print filenames if they're always UTF-8 & can't inc. control chars
```

I will comment on a number of problems that filenames cause the Bourne shell, specifically, because anything that causes problems with Bourne shell scripts interferes with use of Unix/Linux systems. The Bourne shell is *not* going away; it is built into POSIX, it is directly used by nearly every Unix-like system for starting it up, and most GNU/Linux users use Bourne shells for interactive command line use. What's more, the leading contender, C shells (csh), are loathed by many (for an explanation, see "Csh Programming Considered Harmful" by Tom Christiansen). Now, it's true that some issues are innate to the Bourne shell, and cannot be fixed by limiting filenames. The Bourne shell is actually a nice programming language for what it is for, but as noted by Bourne himself, its design requirements led to compromises that can sometimes be irksome. In particular, in most cases Bourne shell scripts will *still* need to double-quote variable references in most cases, even if filenames are limited to more reasonable values. For those who don't know, when using a variable value, you usually need to write `"$file"` and not `$file` in Bourne shells (due to quirks in the language that make it easy to use interactively). You don't need to double-quote values in certain cases (e.g., if they can only contain letters and digits), but those are special cases. Since variables can store information other than filenames, many Bourne shell programmers get into the habit of adding double-quotes around all variables anyway unless they want a special effect, and that effectively resolves the issue. But as shown above, that's not the only issue; it can be difficult to handle all filenames correctly in the Bourne shell *even when* you use double-quotes correctly.

What's more, filename problems tend to happen in *any* language; they are *not* specific to any particular language. For example, if a filename begins with "-", and another command is invoked with that filename as its parameter, that command will see an option flag... no matter *what* computer languages are being used. Similarly, it's more awkward to pass lists of filenames between programs in different languages when newlines can be part of the filename. Practically every language gracefully handles line-at-a-time processing; it'd be nice to be able to easily use that with filenames.

The problem of awkward filenames is so bad that there are programs like detox and Glindra that try to fix "bad" filenames. The POSIX standard includes pathchk; this lets you determine that a filename is bad. But the real problem is that bad filenames were allowed in the first place and aren't prevented or escaped by the system — cleaning them up later is a second-best approach.

2. Programs assume it, standards permit it, Operating Systems already do it

2.1. Programs assume bad filenames won't happen

Lots of programs presume "bad" filenames can't happen, and fail to handle them. For example, many programs fail to handle filenames with newlines in them, because it's harder to write programs that handle such filenames correctly. In several cases, developers have specifically stated that there's no point in supporting such filenames! For example:

- John DuBois stated on 2006-03-27 that "Newlines in filenames are mainly something you would encounter in a malicious context — created by a local user, included in an archive, etc. Since I don't expect to ever see a *useful* file with a newline embedded in its name, and using newline-terminated filenames is often more convenient than using null-terminated filenames, some of the utilities I write simply ignore them".
- As of this writing, bzr can't handle newlines in filenames, and there's no consensus that it even *should*.

- Some versions of CVS fail if filenames have newlines.
- Key security program sha1sum performs odd undocumented behavior when certain odd characters (like newline) are in filenames — here there's an attempt to handle odd filenames, but because it's an undocumented and non-standard result, it can make the problem *worse*.
- A Fedora discussion on "How to iterate over files in a bash script?" discussed this; the author used "while read FILENAME", which has the errors noted above (fails on filenames with newlines, leading/trailing whitespace, or including a backslash).
- Here's another example of people genuinely trying to help each other, again with an incorrect "while read FILENAME" solution.
- Michael Henry's Novalug post "Filename handling: correctness vs. convenience" (2009-02-28) notes that "In the Unix culture, there has been a historical bias against filenames with 'odd' characters... [because they] are harder to deal with at the command line and in scripts... they tend to be avoided in Unix filenames. As a result of this scarcity of unusual filenames, there are often shortcuts that can be taken to avoid the hassle [of] fully general filename handling. These shortcuts truly save a lot of time, and I use them myself frequently; however ... Many of the shortcuts are in essence buggy approximations to a robust general-case solution, but they are so much more convenient than the fully correct solution". I find this text particularly telling; I think that we should modify the filesystem rules, and POSIX tools, so that robust general-case solutions are so *easy* that people will *naturally* create robust, general-case programs.

There are a few programs that *do* try to handle all cases. According to user proski, "One of the reasons git replaced many shell scripts with C code was support for weird file names. C is better at handling them. In absence of such issues, many commands would have remained shell scripts, which are easier to improve". But such exceptions prove the rule — many developers would not be willing to re-write working programs, in a different language, just to handle bad filenames.

Failure to handle "bad" filenames can lead to mysterious failures and even security problems... but only if they can happen at all. If "bad" filenames can't occur, the problems they cause go away too!

2.2. Standards permit the exclusion of bad filenames

The POSIX standard defines what a "portable filename" is; this definition implies that many filenames are *not* portable and thus do not need to be supported by POSIX systems. For all the details, see the Austin Common Standards Revision Group web page. To oversimplify, the POSIX.1-2008 specification is simultaneously released as both The Open Group's Base Specifications Issue 7 and IEEE Std 1003.1(TM)-2008. I'll emphasize the Open Group's version, since it is available at no charge via the Internet (good job!!). Its "base definitions" document section 4.7 ("Filename Portability") says:

> For a filename to be portable across implementations conforming to POSIX.1-2008, it shall consist only of the portable filename character set as defined in Portable Filename Character Set. Portable filenames shall not have the <hyphen> character as the first character since this may cause problems when filenames are passed as command line arguments.

I then examined the Portable Filename Character Set, defined in 3.276 ("Portable Filename Character Set"); this turns out to be just A-Z, a-z, 0-9, <period>, <underscore>, and <hyphen> (aka the dash character). So it's perfectly okay for a POSIX system to reject a non-portable filename due to it having "odd" characters or a leading hyphen.

In fact, the POSIX.1-2008 spec includes a standard shell program called "pathchk", which can be used to determine if a proposed pathname (filename) is portable. Its "-p" option writes a diagnostic if the pathname is too long (more than {_POSIX_PATH_MAX} bytes or contains any component longer than {_POSIX_NAME_MAX} bytes), or contains any character that is not in the portable filename character set. Its “-P” option writes a diagnostic if the pathname is empty or contains a component beginning with a hyphen. GNU, and many others, include pathchk. (My thanks to Ralph Corderoy for reminding me of pathchk.) So not only does the POSIX standard note that some filenames aren’t portable... it even specifically includes tools to help identify bad filenames (such as ones that include control characters or have a leading hyphen in a component).

2.3. Operating Systems already forbid bad filenames in certain cases

Indeed, existing POSIX systems *already* reject some filenames. A common reason is that many POSIX systems mount local or remote filesystems that have additional rules, e.g., for Microsoft Windows. Wikipedia’s entry on Filenames reports on these rules in more detail. For example, the Microsoft Windows kernel forbids the use of characters in range 1-31 (i.e., 0x01-0x1F) in filenames, so any such filenames can’t be shared with Windows users, and they’re not supposed to be stored on their filesystems. I wrote some code and found that the Linux msdos module (which supports one of the Windows filesystems) already rejects some “bad” filenames, returning the EINVAL error message instead.

The Plan 9 operating system was developed by many Unix luminaries; its filenames can only contain printable characters (that is, any character outside hexadecimal 00-1F and 80-9F) and cannot include either slash or blank (per intro(5)). Tom Duff explains why Plan 9 filenames noted that filenames with spaces are a pain for many reasons, in particular, that it messes up scripts. Duff said, “When I was working on the plan 9 shell, I did a survey of all the file names on all the unix machines that I could conveniently look at, and discovered, unsurprisingly, that characters other than letters, digits, underscore, minus, plus and dot were so little used that forbidding them would not impact any important use of the system. Obviously people stick to those characters to avoid colliding with the shell’s syntax characters. I suggested (or at least considered) formalizing the restriction, specifically to make file names easier to find by programs like awk. Probably rob took the more liberal road of forbidding del, space and controls, the first because it is particularly hard to type, and the rest because, as Russ noted, they confound the usual line- and field-breaking rules.”

So some application developers *already* assume that filenames aren’t “unreasonable”, the existing standard (POSIX) *already* permits operating systems to reject certain kinds of filenames, and existing POSIX and POSIX-like systems *already* reject certain filenames in some circumstances. In that case, what kinds of limitations could we add to filenames that would *help* users and software developers?

3. Control characters in filenames are a terrible idea

First: Why the heck are the ASCII control characters (byte values 1 through 31, as well as 127) permitted in filenames? The point of filenames is to create human-readable names for collections of information, but since these characters aren’t readable, the whole point of *having* filenames is lost. There’s no advantage to keeping these as legal characters, and the problems are legion: they can’t be reasonably displayed, many are troublesome to enter (especially in GUIs!), and they cause nothing but nasty side-effects. They also cause portability problems, since filesystems for Microsoft Windows can’t contain bytes 1 through 31 anyway.

One of the nastiest permitted control characters is the newline character. Many programs work a line-at-a-time, with a filename as the content or part of the content; this is great, except it fails when a newline can be in the filename. Many programs simply ignore the problem, and presume that there are no newlines in filenames. But this creates a subtle bug, possibly even a vulnerability — it’d be better to make the no-newline assumption true in the first place! I know of *no* program that legitimately requires the ability to insert newlines in a filename. Indeed, it’s not hard to find comments like “ban newlines in filenames”. GNU’s “find” and “xargs” make it possible to work around this by inserting byte 0 between each filename... but few other programs support this convention (even “ls” normally doesn't, and most shells cannot do word-splitting on \0). Using byte 0 as the separator is a pain to use anyway; who wants to read the intermediate output of this? Even if the only character that is forbidden is newline, that would still help. For example, if newlines can't happen in filenames, you can use a standard (POSIX) feature of xargs (which disables various quoting problems of xargs by escaping each character with a backslash) (lwn forgot the -E option, which I have added):

```
  find . -type f | sed -e 's/./\\&/g' | xargs -E "" somecommand
```

The "tab" character is another control character that makes no sense; if tabs are *never* in filenames, then it's a great character to use as a "column separator" for multi-column data output — especially since many programs *already* use this convention. But the tab character isn't safe to use (easily) if it can be part of a filename.

Some control characters, particularly the escape (ESC) character, can cause all sorts of display problems, including security problems. Terminals (like xterm, gnome-terminal, the Linux console, etc.) implement control sequences. Most software developers don't understand that merely *displaying* filenames can cause security problems if they can contain control characters. The GNU ls program tries to protect users from this effect by default (see the -N option), but many people display filenames without getting filtered by ls — and the problem returns. H. D. Moore's "Terminal Emulator Security Issues" (2003) summarizes some of the security issues; modern terminal emulators try to disable the most dangerous ones, but they can still cause trouble. A filename with embedded control characters can (when displayed) cause function keys to be renamed, set X atoms, change displays in misleading ways, and so on. To counter this, some programs modify control characters (such as find and ls) — making it even harder to correctly handle files with such names.

In any case, filenames with control characters aren't portable. POSIX.1-2008 doesn't include control characters in the "portable filename character set", implying that such filenames aren't portable per the POSIX standard. Wikipedia's entry on Filenames notes that the Windows kernel forbids the use of characters in range 1-31 (i.e., 0x01-0x1F), so any such filenames can't be shared with Windows users, and they're not supposed to be stored on their filesystems.

A few people noted that they used the filesystem as a keystore, and found it handy to use filenames as arbitrary-value keys. That's fine, but filesystems *already* impose naming limitations; you can't use \0 in them, and you can't use '/' as a key value in the same way, even on a traditional Unix filesystem. And as noted above, many filesystems impose more restrictions anyway. So even people who use the filesystem as a keystore, with arbitrary key values, must do *some* kind of encoding of filenames. Since you have to encode anyway, you can use an encoding that is easier to work with and less likely to cause subtle problems... like one that forbids control characters. Many programs, like git, use the filesystem as a keystore yet do not require control characters in filenames.

In contrast, if control characters are forbidden when created and/or escaped when returned, you can safely use control characters like TAB and NEWLINE as filename separators, and the security risks of displaying unfiltered control characters in filenames goes away. As noted above, software developers make these assumptions anyway; it'd be great if it was safe to do so.

4. Leading dashes in filenames are a terrible idea

The "leading dash" (aka leading hyphen) problem is an ancient problem in Unix/Linux/POSIX. This is another example of the general problem that there's interaction between overly-flexible filenames with other system components (particularly option flags and shell scripts).

The Unix-haters handbook page 27 (PDF page 67) notes problems these decisions cause: "By convention, programs accept their options as their first argument, usually preceded by a dash... Finally, Unix filenames can contain most characters, including nonprinting ones. This is flaw #3. These architectural choices interact badly. The shell lists files alphabetically when expanding "*"

[and] the dash (-) comes first in the lexicographic caste system. Therefore, filenames that begin with a dash (-) appear first when "*" is used. These filenames become options to the invoked program, yielding unpredictable, surprising, and dangerous behavior... [e.g., "rm *" will expand filenames beginning with dash, and use those as options to rm]... We've known several people who have made a typo while renaming a file that resulted in a filename that began with a dash: "% mv file1 -file2" Now just try to name it back... Doesn't it seem a little crazy that a filename beginning with a hypen, especially when that dash is the result of a wildcard match, is treated as an option list?" Indeed, people repeatedly ask how to ignore leading dashes in filenames — yes, you can prepend "./", but why do you need to know this at all?"

Similarly, in 1991 Larry Wall (of perl fame) stated: "Just don't create a file called -rf. :-)" in a discussion about the difficulties in handling filenames well.

The list of problems that "leading dash filenames" creates is seemingly endless. You can't safely run "cat *", because there might be a file with a leading dash; if there's a file named "-n", then suddenly all the output is numbered if you use GNU cat. Not all programs support the "--" convention, so you can't simply say "precede all command lists with --", and in any case, people forget to do this in real life. Even the POSIX folks, who are experts, make mistakes due to leading dashes; bug 192 identifies a case where examples in POSIX failed to operate correctly when filenames begin with dash.

You could prefix the name or glob with "./", e.g., "`cat ./*`". Prefixing the filename is a good solution, but people often don't know or forget to do this. The result: many programs break (or are vulnerable) when filenames have components beginning with dash. Users of "find" get this prefixing essentially for free, but then they get troubled by newlines, tabs, and spaces in filenames (as discussed elsewhere).

POSIX.1-2008's "base definitions" document section 4.7 ("Filename Portability") specifically says "Portable filenames shall not have the <hyphen> character as the first character since this may cause problems when filenames are passed as command line arguments". So filenames with leading hyphens are *already* specifically identified as non-portable in the POSIX standard.

There's no reason that a filesystem *must* permit filenames to begin with a dash. If such filenames were forbidden, then writing safe shell scripts would be much simpler — if a parameter begins with a "-", then it's an option and there is no other possibility.

If the filesystem *must* include filenames with leading dashes, one alternative would be to modify underlying tools and libraries so that whenever globbing or directory scanning is done, prepend "./" to any filename beginning with "-". This would be done by glob(3), scandir(3), readdir(3), and shells that implement globbing themselves. Then, "cat *" would become "cat ./-n" if "-n" was in the directory. This would be a silent change that would quietly cause bad code to work correctly. There are reasons to be wary of these kinds of hacks, but if these kinds of filenames must exist, it would at least reduce their trouble. I will say more about solutions later in this paper. Since POSIX says that filename components with leading dashes (hypens) are not portable, you can say that this is all part of a special non-portable extension... and thus meets the POSIX specification.

5. Lack of an encoding standard is a terrible idea — use UTF-8

With today's march towards globalization, computers must support the sharing of information using many different languages. Given that, it's crazy that there's no standard encoding for filenames across all Unix/Linux/POSIX systems. At the beginnings of Unix, everyone assumed that filenames could only be English text, but that hasn't been true for a long time. Yet because you can't know the character encoding of a given filename, in theory you can't display filenames at all today. Why? Because then you don't know how to translate the bytes of a filename into displayable characters (!).

This is true for GUIs, and even for the command line. Yet you *must* be able to display filenames, so you need to make *some* determination... and it will be wrong.

The traditional POSIX approach is to use environment variables that declare the filename character encoding (such as LC_ALL, LC_CTYPE, LC_CTYPE, LC_COLLATE, and LANG). But as soon as you start working with other people (say, by receiving a tarball or sharing a filesystem), the single environment variable approach fails. That's because the single-environment-variable approach assumes that the entire filesystem uses the same encoding (as specified in the environment variable), but once there's file sharing, different parts of the filesystem can use different encoding systems. Should you interpret the bytes in a filename as ISO-8859-1? One of the other ISO-8859-* encodings? KOI8-* (for Cyrillic)? EUC-JP or Shift-JIS (both popular in Japan)? In short, this is too flexible! Since people routinely share information around the world, this incompatibility is awful. The Austin Group even had a discussion about this in 2009. This failure to standardize the encoding leads to confusion, which can lead to mistakes and even vulnerabilities.

Yet this flexibility is actually not flexible enough, because the current filesystem requirements don't permit arbitrary encodings. If you want to store arbitrary international text, you need to use Unicode/ISO-10646. But the other common encodings of Unicode/ISO-10646 (UTF-16 and UTF-32) must be able to store byte 0; since you *can't* use byte 0 in a filename, they don't work at all. The filesystem is also not flexible in another way: There's no mechanism to find out what encoding is used on a given filesystem. If one person uses ISO-8859-1 for a given filename, there's no obvious way to find out what encoding they used. In theory, you could store the encoding system with the filename, and then use multiple system calls to find out what encoding was used for each name.. but really, who *needs* that kind of complexity?!?

If you want to store arbitrary language characters in filenames using todays' Unix/Linux/POSIX filesystem, the *only* widely-used answer that "simply works" for all languages is UTF-8. Wikipedia's UTF-8 entry and Markus Kuhn's UTF-8 and Unicode FAQ have more information about UTF-8. UTF-8 was developed by Unix luminaries Ken Thompson and Rob Pike, specifically to support arbitrary language characters on Unix-like systems, and it's widely acknowledged to have a great design.

When filenames are sent to and from the kernel using UTF-8, then all languages are supported, and there are no encoding interoperability problems. Any other approach would require nonstandard additions like adding sort of "character encoding" value with the filesystem, which would then require user programs to examine and use this encoding value. And they won't. Users and software developers don't need more complexity — they want less. If people simply agreed that "all filenames will be sent in/out of the kernel in UTF-8 format", then all programs would work correctly. In particular, programs could simply retrieve a filename and print it, knowing that the filename is in UTF-8. (Other encodings like UTF-7 and punycode do exist. But these are designed for cases where you can't have byte values more than 127, which is not true for Unix/Linux/POSIX filesystems. Which is why people do not use them for filesystems.) Plan 9 already did this, and showed that you could do this on a POSIX-like system. The IETF specifically mandates that all protocol text must support UTF-8, while all other encodings are optional.

Another advantage of UTF-8 filenames are that they are *very* robust. The chance of a random 4-byte sequence of bytes being valid UTF-8, and not pure ASCII, is only 0.026% — and the chances drop even further as more bytes are added. Thus, systems that use UTF-8 filenames will almost certainly detect when someone tries to import non-ASCII filenames that use the "wrong" encoding — eliminating filename mojibake.

UTF-8 is already supported by practically everything. Some filesystems store filenames in other formats, but at least on Linux, all of them have mount options to translate in/out of UTF-8 for userspace. In fact, some filesystems require a specific encoding on-disk for filenames, but to do this correctly, the kernel has to know which encoding is being used for the data sent in and out (e.g., with iocharset). But not all filesystems can do this conversion, and how do you find out which options are used where?!? Again, the simple answer is “use UTF-8 everywhere”.

There’s also another reason to use UTF-8 in filenames: Normalization. Some symbols have more than one Unicode representation (e.g., a character might be followed by accent 1 then accent 2, or by accent 2 then accent 1). They’d look the same, but they would be considered different when compared byte-for-byte, and there’s more than one normalization system (Programs written for Linux normally use NFC, as recommended by the W3C, but Darwin and MacOS X normally use NFD). If you have a filename in a non-Unicode encoding, then it’s ambiguous how you “should” translate these to Unicode, making simple questions like “is this file already there” tricky. But if you store the name as UTF-8 encoded Unicode, then there’s no trouble; you can just use the filename using whatever normalization convention was used when the file was created (presuming that the on-disk representation also uses some Unicode encoding).

To be fair, what I’m proposing here doesn’t solve some other Unicode issues. Many characters in Unicode look identical to each other, and in many cases there’s more than one way to represent a given character. But these problems *already* exist, and they don’t go away if the status quo continues. If we at least agreed that the userspace filename API was always in UTF-8, we’d at least solve half the battle.

Andrew Tridgell, Samba’s lead developer, has identified yet another reason to use UTF-8 — case handling. Efficiently implementing Windows’ filesystem semantics, where uppercase and lowercase are considered identical, requires that you be able to know what is “uppercase” and what is “lowercase”. This is only practical if you know what the filename encoding is in the first place. (Granted, total upper and lower case handling is in theory locale-specific, but there are ways to address that sensibly that handle the cases people care about... and that’s outside the scope of this article.) Again, a single character encoding system for all filenames, from the application point of view, is almost required to make this efficient.

User “epa” on LWN notes that Python 3 “got tripped up by filenames that are not valid UTF-8”. Python 3 moved to a very clean system where there are “string” types that handle internationalized text and “bytes” that contain arbitrary data. You would think that filenames would be string types, but currently POSIX filenames are really just binary blobs! Python 3’s “what’s new” discusses what they had to do in trying to paper this over, but as epa says, this situation interferes with implementing filenames “as Unicode strings [to] cleanly allow international characters”. Eventually, Python 3.1 implemented the more-complicated PEP 383 proposal, specifically to address the problem that some “character” interfaces (like filenames) don’t just provide characters at all. In PEP 383, on POSIX systems, “Python currently applies the locale’s encoding to convert the byte data to Unicode, failing for characters that cannot be decoded. With this PEP, non-decodable bytes >= 128 will be represented as lone surrogate codes U+DC80..U+DCFF. Bytes below 128 will produce exceptions... To convert non-decodable bytes, a new error handler “surrogateescape” is introduced, which produces these surrogates. On encoding, the error handler converts the surrogate back to the corresponding byte. This error handler will be used in any API that receives or produces file names, command line arguments, or environment variables”.

The result is that many applications end up being *far* more complicated than necessary to deal with the lack of an encoding standard. Python PEP 383 bluntly states that the Unix/Linux/POSIX lack of enforced encoding is a design error: “Microsoft Windows NT has corrected the original design limitation of Unix, and made it explicit in its system interfaces that these data (file names, environment variables, command line arguments) are indeed character data [and not arbitrary bytes]”. Zooko O’Whielacronx posted some comments on Python PEP 383 relating to the Tahoe project. He commented separately to me that “Tahoe could simplify its design and avoid costly storage of ‘which encoding was allegedly used’ next to *every* filename if we instead required utf-8b for all filenames on Linux.” (Sidebar: Tahoe is an interesting project; Here is Zooko smashing a laptop with an axe as part of his Tahoe presentation.)

Converting existing systems or filesystems to UTF-8 isn't that painful either. The program "convmv" can do mass conversions of filenames into UTF-8. This program was designed to be "very handy when one wants to switch over from old 8-bit locales to UTF-8 locales". It's taken years to get some programs converted to support UTF-8, but nowadays almost all modern POSIX systems support UTF-8.

Again, let's look at the POSIX.1-2008 spec. Its "Portable Filename Character Set" (defined in 3.276) is only A-Z, a-z, 0-9, <period>, <underscore>, and <hyphen>. Note that this is a very restrictive list; few international speakers would accept this limited list, since it would mean they must only use English filenames. That's ridiculous; most computer users don't even *know* English. So why is this standard so restrictive? That's *because* there's no standard encoding; since you don't know if a filename is UTF-8 or something else, there's no way to portably share filenames with non-English characters. If we *did* agree that UTF-8 encoding is used, the set of portable characters could include all languages. In other words, the lack of a standard *creates* arbitrary and unreasonable limitations.

Linux distributions are already moving towards storing filenames in UTF-8, for this very reason. Fedora's packaging guidelines require that "filenames that contain non-ASCII characters must be encoded as UTF-8. Since there's no way to note which encoding the filename is in, using the same encoding for all filenames is the best way to ensure users can read the filenames properly." OpenSuSE 9.1 has already switched to using UTF-8 as the default system character set ("lang_LANG.UTF-8"). Ubuntu recommends using UTF-8, saying "A good rule is to choose utf-8 locales", and provides a UTF-8 migration tool as part of its UTF-8 by default feature.

Filename permissiveness is not just a command-line problem. It's actually worse for the GUIs, because if filenames can truly be anything, then GUIs have no way to actually display filenames. The major POSIX GUI suites GNOME and KDE have already moved towards UTF-8 as the required filename encoding format:

1. In a 2003 discussion about GNOME, Michael Meeks noted that "using locale encoded filenames on the disk is a really, really bad idea :-) simply because there is never sufficient information to unwind the encoding (think networking, file sharing, etc.). So — the right way to go is utf-8 everywhere". He noted that although GNOME has an option G_BROKEN_FILENAMES, it is "only a way to help migration towards that. The issue of course is that the whole Unix world needs fixing to be UTF-8 happy..."
2. KDE has this problem, too. They do their best to deal with it by guessing from the user's locale, but they also have the option KDE_UTF8_FILENAMES so that UTF-8-everywhere filesystems are easily handled. This note may be of interest too.

The GUI toolkit Qt (the basis of KDE), since Qt 4, has "removed the hacks they had in QString to allow malformed Unicode data in its QString constructor. What this means is that the old trick of just reading a filename from the OS and making a QString out of it is impossible in general since there are filenames which are not valid ASCII, Latin-1, or UTF-8. Qt does provide a way to convert from the 'local 8-bit' filename-encoding to and from QString, but this depends on there being one, and only one, defined filename-encoding (unless the application wishes to roll its own conversion). This has effectively caused KDE to mandate users use UTF-8 for filenames if they want them to show up in the file manager, be able to be passed around on DBus interfaces, etc."

NFSv4 requires that all filenames be exchanged using UTF-8 over the wire. The NFSv4 specification, RFC 3530, says that filenames should be UTF-8 encoded in section 1.4.3: "In a slight departure, file and directory names are encoded with UTF-8 to deal with the basics of internationalization." The same text is also found in the newer NFS 4.1 RFC (RFC 5661) section 1.7.3. The current Linux NFS client simply passes filenames straight through, without any conversion from the current locale to and from UTF-8. Using non-UTF-8 filenames could be a real problem on a system using a remote NFSv4 system; any NFS server that follows the NFS specification is supposed to reject non-UTF-8 filenames. So if you want to ensure that your files can actually be stored from a Linux client to an NFS server, you *must* currently use UTF-8 filenames. In other words, although some people think that Linux doesn't force a particular character encoding on filenames, in practice it *already* requires UTF-8 encoding for filenames in certain cases.

UTF-8 is a longer-term approach. Systems have to support UTF-8 as well as the many older encodings, giving people time to switch to UTF-8. To use "UTF-8 everywhere", all tools need to be updated to support UTF-8. Years ago, this was a big problem, but as of 2011 this is essentially a solved problem, and I think the trajectory is very clear for those few trailing systems.

Not all byte sequences are legal UTF-8, and you don't want to have to figure out how to display them. If the kernel enforces these restrictions, ensuring that only UTF-8 filenames are allowed, then there's no problem... all the filenames will be legal UTF-8. Markus Kuhn's utf8_check C function can quickly determine if a sequence is valid UTF-8.

The filesystem should be requiring that filenames meet *some* standard, not because of some evil need to control people, but simply so that the names can always be displayed correctly at a later time. The lack of standards makes things *harder* for users, not easier. Yet the filesystem doesn't force filenames to be UTF-8, so it can easily have garbage.

We have a good solution that is already in wide use: UTF-8. So let's use it!

6. Spaces in filenames

6.1. Probably too late for an outright ban on spaces in filenames

It'd be easier and cleaner to write fully-correct shell scripts if filenames couldn't include any kind of whitespace. There's no reason anyone needs tab or newline in filenames, as noted above, so that leaves us with the space character.

There are a *lot* of existing Unix/Linux shell scripts that presume there are no space characters in filenames. Many RPM spec files' shell scripts make this assumption, for example (this can be enforced in their constrained environment, but not in general). Spaces in filenames are particularly a problem because the default setting of the Bourne shell "IFS" variable (which determines how substitution results are split up) includes space as a delimiter. This means that, by default, invoking "find" via '...' or $(...) will fail to handle filenames with spaces (they will break single filenames into multiple filenames at the spaces). Any variable use with a space-containing filename will be split or corrupted if the programmer forgets to surround it with double-quotes (unquoted variable uses can also cause trouble if the filename contains newline, tab, "*", "?", or "]", but these are less common than filenames with spaces). Reading filenames using `read` will also fail (by default) if a filename begins or ends with a space. Many programs, like xargs, also split on spaces by default. The result: Lots of Unix/Linux/POSIX programs don't work correctly on filenames with spaces.

In some dedicated-use systems, you could enforce a "no spaces" rule; this would make some common programming errors no longer an error, reducing slightly the risk of security vulnerabilities. From a functional viewpoint, other characters like "_" could be used instead of space. As noted above, some operating systems like Plan 9 expressly forbid spaces in filenames, so there is even some precedence for having an operating system forbid spaces in filenames.

Unfortunately, a lot of people *do* have filenames with embedded spaces (spaces that are not at the beginning or end of a filename), so a "no spaces" rule would hard to enforce in general. In particular, you essentially *cannot* handle typical Windows and MacOS filenames without handling filenames with an embedded space, because many filenames from those systems use the space character. So if you exchange files with them (via archives, shared storage, and so on), this is often impractical. Also, Windows' equivalent of "/home" is "\Documents and Settings" and Windows' equivalent of "/usr/bin" is "\Program Files" — so you *must* deal with embedded spaces if you deal directly with Windows' primary filesystem from a POSIX system. (Windows Vista and later use "\Users" instead of the awful default "\Documents and Settings", copying the more sensible Unix approach of using short names without spaces, but the problem still remains overall.)

6.2. Banning leading and/or trailing spaces might work

However, there are variations that might be more palatable to many: "no leading spaces" and/or "no trailing spaces". Such filenames are a lot of trouble, especially filenames with trailing spaces — these often confuse users (especially GUI users).

If leading spaces, trailing spaces, newline, and tab can't be in filenames, then a Bourne shell construct already in common use actually becomes correct. A "while" loop using `read -r file` works for filenames if spaces are always between other characters, but by default it subtly fails when filenames have leading or trailing spaces (because space is by default part of the IFS). But if leading spaces, trailing spaces, newline, and tab cannot occur in filenames, the following works all the time with the default value of IFS:

```
# CORRECT IF filenames can't include leading/trailing space, newline, tab,
# even though IFS is left as its default value
find . -print |
while read -r file ; do
  command "$file" ...
done
```

There are a few arguments that leading spaces should be accepted. barryn informs me that "There is a use for leading spaces: They force files to appear earlier than usual in a lexicographic sort. (For instance, a program might create a menu at run time in lexicographic order based on the contents of a directory, or you may want to force a file to appear near the beginning of a listing.) This is especially common in the Mac world....". They are even used by some people with Mac OS X.

But it's hard to argue that trailing spaces are useful. Trailing spaces are worse than leading ones; in many user interfaces, a leading space will at least cause a visible indent, but there's no indication at all of trailing spaces... leading to rampant confusion. I understand that in Microsoft Windows (or at least some of its key components), the space (and the period) are not allowed as the final character of a filename. So preventing a space as a final character improves portability, and is rather unlikely to be required for interoperability.

If trailing spaces are forbidden, then filenames with *only* spaces in them become forbidden as well. And that's a good thing; filenames with *only* spaces in them are *really* confusing to users. Years ago my co-workers set up a directory full of filenames with only spaces in them, briefly stumping our Sun representative.

So banning trailing spaces in a component might be a plausible broad rule. It's not as important as getting rid of newlines in filenames, but it's worth considering, because it would get rid of some confusion. Banning both leading and trailing spaces is also plausible; doing so would make `while read -r` correct in Bourne shell scripts.

6.3. Interesting alternative: Auto-convert spaces to unbreakable spaces

James K. Lowden proposed an interesting alternative for spaces: "Spaces could be transparently handled (no pun intended) with U+00A0, a non-breaking space, which in fact it is. Really. If the system is presented with a filename containing U+0020, it could just replace it unilaterally with the non-breaking space [Unicode U+00A0, represented in UTF-8 by the hex sequence 0xC2 0xA0]. Permanently, no questions asked."

This idea is interesting, because by default Bourne shells only break on U+0020, so they would consider the filename as one long unbreakable string. Filenames really aren't intended to be broken up, so that's actually a defensible representation. He claims "For most purposes, that will be just fine. GUIs won't mind. Shells won't mind; most scripts will be happier."

He does note that constructs like

```
if [ "$name" = "my nice name" ]
```

will fail, but he and I suspect that such code is rare. He says, "scripts won't typically contain hard-coded comparisons to filenames with spaces".

I'm guessing that the *filesystem* would internally always store spaces, but the *API* would always get unbreakable spaces. This could cause problems if other systems stored filenames on directories which only differed between the use of unbreakable spaces and regular spaces, but users would generally think that's pretty evil in the first place.

I'm not sure how I feel about this one idea, but it's certainly an interesting approach that's worth thinking about. One reason I hesitate is that if other things are fixed, the difficulties of handling spaces in filenames diminishes anyway, as I'll explain next.

One reader of this essay suggested that GUIs should transparently convert spaces to underscores when creating a file, reversing this when displaying a filename. It's an interesting idea. However, I fear that some evil person will create multiple files in one directory which only differ because one uses spaces and the other uses underscores. That might look okay, but would create opportunity for confusion in the future. Thus, I haven't recommended this apporoach.

6.4. Spaces in filenames are less bad if other problems fixed

Having spaces in filenames is no disaster, though, particularly if other problems are fixed.

First, it's worth noting that many "obvious" shell programs already work correctly, today, even if filenames have spaces and you make no special settings. For example, glob expansions like "`cat ./*`" works correctly, even if some filenames have spaces, because file glob expansion occurs *after* splitting (more about this in a moment). The POSIX specification specifically requires this, and this is implemented correctly by lots of shells (I've checked bash, dash, zsh, ksh, and even busybox's shell). The find commands's "-exec" option can work with arbitrary filenames (even ones with control characters), though I find that if the exec command gets long, the script starts to get very confusing:

```
# This is straightforward:
find . -type f -exec somecommand {} \;
# As these get long, I scream (example from "explodingferret"):
find . -type f -exec sh -c 'if true; then somecommand "$1"; fi' -- {} \;
```

Once newlines and tabs *cannot* happen in filenames, programs can safely use newlines and tabs as delimiters between filenames. Having safe delimiters makes spaces in filenames much easier to handle. In particular, programs can then *safely* do what many *already* do: they can use programs like 'find' to create a list of filenames (one per line), and then process the filenames a line at a time.

However, if we stopped here, spaces in filenames still cause problems for Bourne shell scripts. If you invoke programs like find via command substitution, such as "``for file in `find .` ``", then by default the shell will break up filenames on the spaces — corrupting the results. This is one of the reasons that many shell scripts don't handle spaces-in-files correctly. Yet the "obvious" way to process files is to create a loop through the results of a command substitution with `find`! We can make it *much* easier to write correct shell scripts by using a poorly-documented trick.

6.5. A trick: Change IFS to just newline and tab

Writers of (Bourne-like) shell scripts can use an additional trick to make spaces-in-filenames easier to handle, as long as newlines and tabs can't be in filenames. The trick: set the "IFS" variable to be just newline and tab.

6.5.1. What is IFS?

IFS (the "input field separator") is an ancient, very standard, but not well-known capability of Bourne shells. After almost all substitutions, including command substitution '...' and variable substitution ${...}, the characters in IFS are used to split up any substitution results into multiple values (unless the results are inside double-quotes). Normally, IFS is set to space, tab, and newline — which means that by default, after almost all substitutions, spaces are interpreted as separating the substituted values into different values. This default IFS setting is very bad if file lists are produced through substitutions like command substitution and variable substitution, because filenames with spaces will get split into multiple filenames at the spaces (oops!). And processing filenames is *really common*.

Changing the IFS variable to include only newline and tab makes lists of filenames *much* easier to deal with, because then filenames with spaces are trivially handled. Once you set IFS this way, instead of having to create a "while read..." loop, you can place a '...' file-listing command in the "usual place" of a file list, and filenames with spaces will then work correctly. And if filenames can't include tabs and newlines, you can correctly handle *all* filenames.

A quick clarification, if you're not familiar with IFS: Even when the space character is removed from IFS, you can still use space in shell scripts as a separator in commands or the 'in' part of for loops. IFS only affects the splitting of unquoted values that are *substituted* by the shell. So you can still do this, even when IFS doesn't include space:

```
for name in one two three ; do
  echo "$name"
done
```

6.5.2. How to change IFS to just newline and tab

I recommend using this portable construct near the beginning of your (Bourne-like) shell scripts:

```
IFS="`printf '\n\t'`"
```

If you have a really old system that doesn't include the POSIX-required printf(1), you could use this instead (my thanks to Ralph Corderoy for pointing out this issue, though I've tweaked his solution somewhat):

```
IFS="`echo nt | tr nt '\012\011'`"
```

It's quite plausible to imagine that in the future, the standard "prologue" of a shell script would be:

```
#!/bin/sh
set -eu
IFS="`printf '\n\t'`"
```

An older version of this paper suggested setting IFS to tab followed by newline. Unfortunately, it can be slightly awkward to set IFS to just tab and newline, in that order, using only standard POSIX shell capabilities. The problem is that when you do command substitution in the shell with '...' or

$(...), trailing newline characters are removed *before* the result is used (see POSIX shell & utilities, section 2.6.3). Removing trailing newlines is almost always what you want, but not if the last character you wanted *is* newline. You can also include a newline in a variable by starting a quote and inserting a newline directly, but this is easy to screw up; any other white space could be silently inserted there, including text-transformation tools that might insert \r\n at the end, and people might "help" by indenting your code and quietly ruining it. There's also the problem that the POSIX standard's "echo" is almost featureless, but you can just use "printf" instead. In an older version of this paper I suggested doing `IFS="`printf '\t\nX'`" ; IFS="${IFS%X}"` However, On LWN.net, Explodingferret pointed out a much better portable approach — just reverse their order. This doesn't have the *exactly* the same result as my original approach (parameters are now joined by newline instead of tab when they are joined), but I think it's actually slightly better, and it's definitely simpler. I thought his actual code was harder to read, so I tweaked it (as shown above) to make it clearer.

A slightly more pleasant approach in Bourne-like shells is to use the `$'...'` extension. This isn't standard, but it's widely supported, including by the bash, ksh (korn shell), and zsh shells. In these shells you can just say `IFS=$'\n\t'` and you're done, which is slightly more pleasant. As the korn shell documentation says, the purpose of `'...'` is to 'solve the problem of entering special characters in scripts [using] ANSI-C rules to translate the string... It would have been cleaner to have all "..." strings handle ANSI-C escapes, but that would not be backwards compatible.' It might even be more efficient; some shells might implement 'printf ...' by invoking a separate process, which would have nontrivial overhead (shells can optimize this away, too, since printf is typically a builtin). But this `$'...'` extension isn't supported by some Bourne-like shells, including dash (the default /bin/sh in Ubuntu) and the busybox shell, and the portable version isn't *too* bad. I'd like to see `$'...'` added to a future POSIX standard and these other shells, as it's a widely implemented and useful extension. I think `$'...'` will in the next version of the POSIX specification (you can blame me for proposing it).

6.5.3. Writing shell scripts with IFS set to newline and tab

If filenames can't include newline or tab, and IFS is set to just newline and tab, you can safely do this kind of thing to correctly handle all filenames:

```
for file in `find . -type f` ; do
  some_command "$file" ...
done
```

This `for` loop is a better construct for file-at-a-time processing than the `while read -r file` construct listed earlier. This `for` loop isn't in a separate subprocess, so you can set variables inside the loop and have their values persist outside the loop. The for loop has direct, easy access to standard input (the while loop uses standard input for the list of filenames). It's shorter and easier to understand, and it's less likely to go wrong (it's easy to forget the "-r" option to read).

Some people like to build up a sequence of options and filenames in a variable, using the space character as the separator, and then call a program later with all the options and filenames built up. That general approach still works, but if the space character is not in IFS, then you can't easily use it as a separator. Nor should you — if filenames can contain spaces, then you must *not* use the space as a separator. The solution is trivial; just use newlines or tabs as the separator instead. The usual shell tricks still apply (for example, if variable x leads with separators, then $x without quotes will cause the variable to get split using IFS and the leading separators will be thrown away). This is easiest to show by example:

```
# DO NOT DO THIS when the space character is NOT part of IFS:
  x="-option1 -option2 filename1"
  x="$x filename2"  # Build up $x
```

```
    run_command $x
  # Do this instead:
    t=`printf "\t"` # Newline is tricky to portably set; use tab as separator
    x="-option1${t}-option2${t}filename1"
    x="$x${t}filename2"  # Build up $x.
    run_command $x
  # Or do this (do NOT give printf a leading dash, that's not portable):
    x=`printf "%s\n%s\n%s" "-option1" "-option2" "filename1"`
    x=`printf "%s\n%s" "$x" "filename2"`  # Build up $x.
    run_command $x
```

Do not use plain “read” in Bourne shells &mdash use “read -r”. This is true regardless of the IFS setting. The problem is that “read”, when it sees a backslash, will merge the line with the next line, unless you undo that with “-r”. Notice that once you remove space from IFS, read stops corrupting filenames with spaces, but you still need to use the -r option with read to correctly handle backslash.

Of course, there are times when it’s handy to have IFS set to a different value, including its traditional default value. One solution is straightforward: Set IFS to the value you need, when you need it... that’s what it’s there for. So feel free to do this when appropriate:

```
#!/bin/sh
set -eu
traditionalIFS="$IFS"
IFS="`printf '\n\t'`"
...
IFS="$traditionalIFS"
# WARNING: You usually want "read -r", not plain "read":
while read -r a b c
do
  echo "a=$a, b=$b, c=$c"
done
IFS="`printf '\n\t'`"
```

Setting IFS to a value that ends in newline is a little tricky. If you just want to temporarily restore IFS to its default value, just save its original value for use it later (as shown above). If you need IFS set to some other value with newline at the end, this kind of sequence does the trick:

```
 IFS="`printf '\t\nX'`"
 IFS="${IFS%X}"
```

Setting IFS to newline and tab is best if programs use newline or tab (not space) as their default data separator. If the data format is under your control, you could change the format to use newline or tab as the separator. It turns out that many programs (like GNU seq) already use these separators anyway, and the POSIX definition of IFS makes this essentially automatic for built-in shell commands (the first character of IFS is used as the separator for variables like $*). Once IFS is reset like this, filenames with spaces become *much* simpler to handle.

7. Metacharacters in filenames are unfortunate

Characters that must be escaped in a shell before they can be used as an ordinary character are termed “shell metacharacters”. If filenames cannot contain some or all shell metacharacters, then some security vulnerabilities due to programming errors would go away.

I doubt all POSIX systems would forbid shell metacharacters, but it’d be nice if administrators could configure *specific* systems to prevent such filenames on higher-value systems, as sort of a belt-and-suspenders approach to counter errors in important programs. Many systems are dedicated to specific tasks; on such systems, a filename with unusual characters can only occur as part of an attack.

To make this possible, software on such systems must not *require* that filenames have metacharacters, but that's almost never a problem: Filenames with shell metacharacters are very rare, and these characters aren't part of the POSIX portable filename character set anyway.

Here I'll discuss a few options. One option is to just forbid the glob characters (*, ?, and [) — this can eliminate many errors due to forgetting to double-quote a variable reference in the Bourne shell. You could forbid the XML/HTML special characters "<", ">", "&", and """, which would eliminate many errors caused by incorrectly escaping filenames. You could forbid the backslash character — this would eliminate a less-common error (forgetting the -r option of Bourne shell read). Finally, you could forbid all or nearly all shell meta-characters, which can eliminate errors due to failing to escape metacharacters where required in many circumstances.

7.1. Forbid shell glob characters ("*", "?", and "[") on some systems (so forgetting double-quotes is a non-problem)

All the Bourne shell programming books tell you that you're supposed to double-quote all references to variables with filenames, e.g., `cat "$file"`. Without special filesystem rules, you definitely need to! In fact, correctly-written shell programs must be absolutely infested with double-quotes, since they have to surround almost every variable use. But I find that real people (even smart ones!) make mistakes and sometimes fail to include those quotation marks... leading to nasty bugs.

Although shell programming books don't note it, you can actually omit the double quotes around variable references containing filenames if (1) IFS contains only newline and tab (not a space, as discussed above), and (2) tab, newline, and the shell globbing metacharacters (namely "*", "?", and "[") can't be in the filename. (The other shell metacharacters don't matter, due to the POSIX-specified substitution order of Bourne shells.) This means that `cat $file` would work correctly in such cases, even if `$file` contains a space and other shell metacharacters. From a shell programming point of view, it'd be neat if such control and globbing characters could never show up in filenames... then correct shell scripts could be much cleaner (they wouldn't require all that quoting).

I doubt there can be widespread agreement on forbidding all the globbing metacharacters across *all* Unix-like systems But if local systems reject or rename such names, then when someone accidentally forgets to quote a variable reference with a filename (it happens all the time), the the error cannot actually cause a problem. And that's a great thing, especially for high-value servers (where you could impose more stringent naming rules). Older versions of this article mistakenly omitted the glob character issues; my thanks to explodingferret for correcting that. Similarly, if you also forbid spaces in filenames, as well as these other characters, then even without changing IFS, scripts which accidentally didn't double-quote the variables would still work correctly. (Even if glob metacharacters can be in filenames, there are still good reasons to remove the space character from IFS, as noted in the section on spaces in filenames.)

So, by forbidding a few more characters — at least locally on high-value systems — you eliminate a whole class of programming errors that sometimes become security vulnerabilities. You will still need to put double-quotes around variables that contain values other than filenames, so this doesn't eliminate the general need to surround variables with double-quotes in Bourne-like shells. But by forbidding certain characters in filenames, you decrease the likelihood that a common programming error can turn into an attack; in some cases that's worth it.

7.2. Forbid XML/HTML special characters: <, >, "

You could forbid the XML/HTML special characters "<", ">", "&", and """, which would eliminate many errors caused by incorrectly escaping filenames for XML/HTML.

This would also get rid of some nasty side-effects for shell and Perl programs. The < and >

symbols redirect file writes, for both shell and Perl. This can be especially nasty for Perl, where filenames that begin with < or > can cause side-effects when open()ed — see "man perlopentut" for more information. Indeed, if you use Perl, see "man perlopentut" for other gotchas when opening files in Perl.

7.3. Forbid backslash character

You could forbid the backslash character. This would eliminate one error — forgetting the -r option of Bourne shell `read`.

7.4. Forbid all shell metacharacters on some systems

Of course, you could go further forbid all (or nearly all) shell metacharacters.

Sometimes it's useful to write out programs and run them later. For example, shell programs can be flattened into single long strings. Although filenames are *supposed* to be escaped if they have unusual characters, it's not at all unusual for a program to fail to escape something correctly. If filenames never had characters that needed to be escaped, there'd be one less operation that could fail.

A useful starting-point list is "**`*?:[]"<>|(){}&'!\;`**" (this is Glindra's "safe" list with ampersand, single-quote, bang, backslash, and semicolon added). The colon causes trouble with Windows and MacOS systems, and although opening such a filename isn't a problem on most Unix/Linux systems, the colon causes problems because it's a directory separator in many directory or file lists (including PATH, bash CDPATH, gcc COMPILER_PATH, and gcc LIBRARY_PATH), *and* it has a special meaning in a URL/URI. Note that < and > and & and " are on the list; this eliminates many HTML/XML problems! I'd need to go through a complete analysis of all characters for a final list; for security, you want to identify everything that is permissible, and disallow everything else, but its manifestation can be either way as long as you've considered all possible cases.

In fact, for portability's sake, you already don't want to create filenames with weird characters either. MacOS and Windows XP also forbid certain characters/names. Some MacOS filesystems and interfaces forbid ":" in a name (it's the directory separator). Microsoft Windows' Explorer interface won't let you begin filenames with a space or dot, and Windows also restricts these characters:

```
: * ? " < > |
```

Also, in Windows, \ and / are both interpreted as directory name separators, and according to that page there are some issues with ".", "[", "]", ";", "=", and ",".

For more info, see Wikipedia's entry on Filenames. Windows' NTFS rules are actually complicated, according to Wikipedia:

> Windows kernel forbids the use of characters in range 1-31 (i.e., 0x01-0x1F) and characters `" * : < > ? \ / |`. Although NTFS allows each path component (directory or filename) to be 255 characters long and paths up to about 32767 characters long, the Windows kernel only supports paths up to 259 characters long. Additionally, Windows forbids the use of the MS-DOS device names AUX, CLOCK$, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, CON, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, LPT9, NUL and PRN, as well as these names with any extension (for example, AUX.txt), except when using Long UNC paths (ex. \\.\C:\nul.txt or \\?\D:\aux\con). (In fact, CLOCK$ may be used if an extension is provided.) These restrictions only apply to Windows — Linux, for example, allows use of `" * : < > ? \ |` even in NTFS [The source also included "/" in the last list, but Wheeler believes that is incorrect and has removed it.]

Microsoft Windows also makes some terrible mistakes with its filesystem naming; the section on Windows filename problems briefly discusses this.

8. Beware other assumptions about filenames

Beware of other assumptions about filenames. In particular, filenames that appear different may be considered the same by the operating system, particularly on Mac OS X, Windows, and remote filesystems (e.g., via NFS).

The git developers fixed a critical vulnerability in late 2014 (CVE-2014-9390) due to filenames. GitHub has an interesting post about it. Mercurial had the same problem (they notified the git developers about it). In particular, filenames that appear different are considered the same:

1. On many systems (including Mac OS X, Windows, and many remotely-accessible filesystems accessible via NFS), upper and lower case are not considered distinct. So ".Git" or ".GIT" are considered the same as ".git".
2. In addition, the Mac OS X HFS+ file system considers certain Unicode codepoints as ignorable. For example, committing e.g. .g\u200cit/config and then pulling it on HFS+ would overwrite .git/config because U+200C is one of those ignorable codepoint. In general, Mac OS X normalizes filenames, and in many circumstances considers "different" filenames the same in many cases.

Thus, filtering based on filenames is tricky and potentially dangerous. This is in *addition* to the Windows-specific filenames (e.g., NUL) as discussed above.

Microsoft Windows has a whole host of other nasty tricks involving filenames. Normally periods and spaces at the end of a filename are silently stripped, e.g., "hello .. " is the same filename as "hello". You can also add various other selectors, e.g., "file1::$DATA" is the same as "file1", but the stripping does not happen so "file1...::$DATA" is not the same as "file1". Short 8+3 filenames can refer to longer names. There are other issues too, but this is not primarily an essay about Windows filenames; I just thought it important to note. ???

9. Other shell tricks

There are lots of tricks we caN use in Bourne-like shells to work correctly, or at least not fail catastrophically, with nasty filenames. We’ve already noted a key approach: Set IFS early in a script to prevent breaking up filenames-with-spaces in the wrong place:

```
IFS="`printf '\n\t'`"
```

The problem has been around for a long time, and I can’t possibly catalog all the techniques. Indeed, that’s the problem; we need too many techniques.

I guess I should mention a few other techniques for either handling arbitrary filenames, or filtering out “bad” filenames. I think they’ll show why people often don’t do it “correctly” in the first place. In Bourne shell, you must double-quote variable references for many other kinds of variables anyway, so let’s look beyond that. I will focus on using shell globbing and “find”, since those are where filenames often come from, and the ways for doing it aren’t always obvious. This BashFAQ answer gives some suggestions, indeed, there’s a lot of stuff out there on how to work around these misfeatures.

9.1. Globbing

Shell globbing is great when you just want to look at a list of files in a specific directory and ignore its “hidden” files (files beginning with “.”), particularly if you just want ones with a specific extension. Globbing doesn't let you easily recurse down a tree of files, though; for that, use "file" (below). Problem is, globs happily return filenames that begin with a dash.

When globbing, make sure that your globs cannot return anything beginning with "-", for example, prefix globs with "./" if they start in the current directory. This eliminates the "leading dash" problem in a simple and clean way. Of course, this only works on POSIX; if you can get Windows filenames of the form C:\Users, you'll need to consider drive: as well. When you glob using this pattern, you will quietly hide any leading dashes, skip hidden files (as expected), and you can use any filename (even with control characters and other junk):

```
for file in ./*.jpg ; do ... command "$file"
```

When globbing, make sure that your globs cannot return anything beginning with "-", for example, prefix globs with "./" if they start in the current directory. This eliminates the "leading dash" problem in a simple and clean way.

Making globbing safe for all filenames is actually not complicated — just prefix them with "./". Problem is, nobody knows (or remembers) to prefix globs with "./", leading to widespread problems with filenames starting with "-". If we can't even get people to do *that* simple prefixing task, then expecting them to do complicated things with "find" is silly.

Bash has an extension that can limit filenames, GLOBIGNORE, though setting it to completely deal with all these cases (while still being usable) is a *very* tricky. Here's a GLOBIGNORE pattern so that globs will ignore filenames with control characters, leading dashes, or begin with a ".", as well as traditional hidden files (names beginning with "."), yet accept reasonable patterns (including those beginning with "./" and "../" and even multiple "../"):

```
GLOBIGNORE=`printf '.[!/.]*:..[!/]*:*/.[!/.]*:*/..[!/]*:*[\001-\037\177]*:-*'`
```

By the way, a special thanks to Eric Wald for this complicated GLOBIGNORE pattern, which resolves the GLOBIGNORE problems I mentioned in earlier versions of this article. With this pattern, if you *remember* to always prefix globs with "./" or similar (as you should), then you'll safely get filenames that begin with dash (because they will appear as "./-NAME"). But when you forget to correctly prefix globs (and you will), then leading-dash filenames will be skipped (which isn't ideal, but it's generally far safer than silently changing command options). Yes, this GLOBIGNORE pattern is hideously complicated, but that's my point: Safely traversing filenames is difficult, and it should be easy.

Globbing can't express UTF-8, so you can't filter out non-UTF-8 filenames with globbing. Again, you probably need a separate program to filter out those filenames.

9.2. Find

How can we use `find` correctly? Thankfully, "find" always prefixes filenames with its first parameter, so as long as the first parameter doesn't begin with a dash (it's often "."), we don't have the "leading dash" problem. (If you're starting from a directory that begins with "-" inside your current directory, you can always prefix its name with "./").

It's worth noting that if you want to handle fully-arbitrary filenames, use "find . ... -exec" when you can; that's 100% portable, and can handle arbitrarily-awkward filenames. The more-recent POSIX addition to find of -exec ... {} + can help too. So where you can, do this kind of thing:

```
# This is correct and portable; painful if "command" gets long:
find . ... -exec command {} ;
# This is correct and standard; some systems don't implement this:
find . ... -exec command {} +
```

When you can't do that, using find ... -print0 | xargs -0 is the common suggestion; that works, but those require non-standard extensions (though they are common), the resulting program can get really clumsy if what you want to do if the file isn't simple, and the results don't easily feed into shell command substitutions if you plan to pass in \0-separated results.

If you don't mind using bash extensions, here's one of the better ways to directly implement a shell loop that takes "find"-created filenames. In short, you use a while loop with 'read' and have read delimit only on the \0 (the IFS= setting is needed or filenames containing leading/trailing IFS characters will get corrupted; the `-d ''` option switches to \0 as the separator, and the `-r` option disables backslash processing). Here's a way that at least works in simple cases:

```
# This handles all filenames, but uses bash-specific extensions:
find . -print0 |
while IFS="" read -r -d "" file ; do ...
  # Use "$file" not $file everywhere.
done
```

This approach does handle all filenames, but because we use a pipe, each of the processes will be in a subshell. Thus, if any variables are set inside the "while" loop, their values will disappear once we exit the loop (because the loop's subshell will disappear). To solve *that* problem, we'll need to use *another* nonstandard bash extension, process substitution (which even doesn't work on all systems with bash):

```
# This handles all filenames, but uses bash-specific extensions:
while IFS="" read -r -d "" file ; do ...
  # Use "$file" not $file everywhere.
  # You can set variables, and they'll stay set.
done < <(find . -print0)
```

We can now loop through all the filenames, and retain any variable values we set, but this construct is hideously ugly and non-portable. Also, this approach means we can't read the original standard input, which in many programs would be a problem. You can work around that by using other file descriptors, but that causes even more complications, leading to hideous results. Is there any wonder nobody actually *does* this correctly?!?

Notice that you can't portably use this construct in "for" loops or as a command substitution, due to limitations in current shells (you can't portably say "split input on \0").

Oh, and while carefully using the find command can *process* filenames with embedded control characters (like newline and escape), what happens afterwords that can be "interesting". In GNU find, if you use -print (directly or implicitly) to a teletype, it will silently change the filenames to prevent some attacks and problems. But once piped, there's no way to distinguish between filenames-with-newlines and newlines-between-filenames (without additional options like the nonstandard -print0). And those later commands must be careful; merely *printing* a filename via those later commands is dangerous (since it may have terminal escape codes) and can go badly wrong (because the filename encoding need not match the environment variable settings).

Can you use the 'find' command in a portable way so it will filter out bad filenames, and have a simpler life from there on? Yes! If you have to write secure programs on systems with potentially bad filenames, this may be the way to go — by filtering out the bad filenames, you at least prevent your program from getting affected by them. Here's the simplest portable (POSIX-compliant) approach I've found which filters out filenames with embedded ASCII control characters (including newline and tab); that way, newlines can separate filenames, displaying filenames is less dangerous (though we still have character encoding issues), and the results are easy to use in a command substitution (including a Bourne shell "for" loop) and with line-processing filters:

```
# This is correct and portable; it skips filenames with control chars:
IFS="`printf '\n\t'`" # Remove spaces so spaces-in-filenames still work
```

```
  controlchars=`printf '*[\001-\037\177]*'`
  for file in `find . ! -name "$controlchars"'` ; do
    command "$file" ...
  done
```

Unfortunately, UTF-8 can't really be expressed with traditional globs, because globs can't express a repetition of particular patterns. The standard find only supports globs, so it can't do utf-8 matching by itself. In the long term, I hope "find" grows a simple option to determine if a filename is UTF-8. Full regular expressions *are* able to represent UTF-8, thankfully. So in the short term, if you want to only accept filenames that are UTF-8, you'll need to filter the filename list through a regex (rejecting names that fail to meet UTF-8 requirements). (GNU find has "-regex" as an extension, which could do this, but obviously that wouldn't port to other implementations of find.) Or you could write a small C program that filters them out (along with other bad patterns).

Of course, if filenames are clean (at least, can't have control characters), this can become this far simpler, and that's the point of this article:

```
  IFS="`printf '\n\t'`" # Remove spaces so spaces-in-filenames will work
  ...
  # This is correct if filenames can't have control characters:
  for file in `find .` ; do ...
  done
  # This will fail if scaled to very large lists, but it is correct for
  # smaller lists if filenames can't have control characters:
  cat `find . -type f`
```

9.3. Why do I need to keep using tricks?

Why do I need to add odd coding mechanisms that say "don't send me garbage", and constantly work around the garbage other programs copy to me? There are many conventions out there to try to deal with garbage, but it's just too easy to write programs that fail to do so. Shouldn't the system keep out the garbage in the first place?!?

Yes, I need to filter inputs provided by untrusted programs. Fine. But the operating system kernel shouldn't be one of the untrusted programs I must protect myself against (grin).

Using the techniques discussed above, you can count how many filenames include control characters 1-31 or 127 in the entire system's filesystem:

```
  badfile=`printf '*[\\x01-\\x1f\\x7f]*'`
  find / -name "$badfile" -exec echo 1 \; | wc -l
```

For most systems, the answer is "0". Which means this capability to store weird filenames isn't really necessary. This "capability" costs a lot of development time, and causes many bugs; yet in return we get no real benefit.

10. It works!

So does limiting filenames, even in small ways, actually make things better? Yes! Let me focus on eliminating control characters (at least newline and tab), probably the worst offenders, and how things like a better IFS setting can improve things in a very public historical complaint about Unix.

The Unix-haters handbook page 167 (PDF page 205) begins Jamie Zawinski's multi-page description of his frustrated 1992 effort to simply "find all .el files in a directory tree that didn't have a corresponding .elc file. That should be easy." After much agony (described over multiple pages), he found that the "perversity of the task had pulled me in, preying on my morbid fascination". He ended up writing this horror, which is both horribly complicated and *still* doesn't correctly handle all filenames:

```
find . -name '*.el' -print \
| sed 's/^/FOO=/' | \
sed 's/$/; if [ ! -f \ ${FOO}c ]; then \
echo \ $FOO ; fi/' | sh
```

Zawinski's script fails when filenames have spaces, tabs, or newlines. In fact, just about *any* shell metacharacter in a filename will cause *catastrophic* effects, because they will be executed (unescaped!) by another shell.

Paul Dunne's review of the "Unix Hater's Handbook" (here and here) proposes a solution, but his solution is both *wrong* and *complicated*. Dunne's solution is wrong because it only examines the directories that are the immediate children of the current directory; it fails to examine the current directory and it fails to examine deeper directories. Whups! In addition, his solution is quite complicated; he uses a loop inside another loop to do it, and has to show it in steps (presumably because it's too complicated to show at once). Dunne's solution *also* fails to handle filenames with spaces in them, and it even fails if there are empty directories. Dunne does note those last two weaknesses, to be fair. Dunne doesn't even show the full, actual code; he only shows a code outline, and you have to fill in the pieces before it would actually run. (If it's so complicated that you can only show an outline, it's too complicated.) This is all part of the problem — if it's too hard to write good examples of easy tasks that do the job correctly, then the system is making it too hard to do the job correctly!

Here's my alternative; this one is simple, clear, and actually correct:

```
# This is correct if filenames can't include control characters:
IFS="`printf '\n\t'`"
for file in `find . -name '*.el'` ; do
  if [ ! -f "${file}c" ] ; then
    echo "$file"
  fi
done
```

This approach (above) just sets IFS to the value it should normally have anyway, followed by a single bog-standard loop over the result of "find". This alternative is *much* simpler and clearer than either solutions, it actually handles the entire tree as Zawinski wanted (unlike Dunne's), *and* it handles spaces-in-filenames correctly (as neither of the above do). It also handles empty directories, which Dunne's doesn't, and it handles metacharacters in filenames, which Zawinski's doesn't. It works on *all* filenames (including those with spaces), presuming that filenames can't contain control characters. The find loop presumes that filenames cannot include newline or tab; the later "echo" that prints the filename presumes that the filename cannot contain characters (since if it did, the echo of control characters might cause a security vulnerability). If we also required that filenames be UTF-8, then we could be certain that the displayed characters would be sensible instead of mojibake. This particular program works even when file components begin with "-", because "find" will prefix the filenames with "./", but preventing such filenames is still a good idea for many other programs (the call to `echo` would fail and possibly be dangerous if the filename had been acquired via a glob like `*`). My approach also avoids piping its results to another shell to run, something that Zawinski's approach does. There's nothing wrong with having a shell run a program generated by another program (it's a powerful technique), but if you use this technique, small errors can have catastrophic effects (in Zawinski's example, a filename with metacharacters could cause disaster). So it's best to use the "run generated code" approach only when necessary. This is a trivial problem; such powerful grenade-like techniques should *not* necessary! Most importantly, it's easy to generalize this approach to arbitrary file processing.

That's my point: ***Adding small limits to filenames makes it much easier to create completely-correct programs***. Especially since most software developers act as if these limitations were already being enforced.

Adding small limits to filenames makes it much easier to create completely-correct programs.

Peter Moulder sent me a shorter solution for this particular problem (he accidentally omitted -print, which I added):

```
# Works on all filenames, but requires a non-standard extension, and there
# are security problems with some versions of find when printing filenames:
find . -name '*.el' \! -exec test -e '{}c' \; -print
```

However, Moulder's solution uses an implementation-defined (non-standard) extension; as noted by the Single UNIX specification version 3 section on find, "If a utility_name or argument string contains the two characters "{}", but not just the two characters "{}", it is implementation-defined whether find replaces those two characters or uses the string without change". My thanks to Davide Brini who pointed out that this is implementation-defined, and also suggested this standard-conforming solution instead:

```
# This is correct for all filenames, and portable, but hideously ugly; it can
# cause security vulnerabilities b/c it prints filenames with control chars:
find . -name "*.el" -exec sh -c '[ ! -f "$1"c ] && printf "%s\n" "$1"' sh {} \;
```

This version (with find) can process files with newlines, but if files have embedded newlines, the output is ambiguous. In addition, if the files can have terminal escapes or a different character encoding, *beware* — this code is a security vulnerability waiting to happen. In any case, as file processing gets more complicated, stuffing logic into "find" gets very painful. I believe that the simple for-loop is easier to understand and more easily scales to more complicated file processing.

Similarly, here is a little script called mklowercase, which renames all filenames to lowercase recursively from the current directory (".") down. Again, this script is pretty simple to write *if* we can assume that filenames don't include newline or tab. This one can handle filenames with spaces and initial dash (again, because find can handle them):

```
#!/bin/sh
# mklowercase - change all filenames to lowercase recursively from "." down.
# Will prompt if there's an existing file of that name (mv -i)

set -eu
IFS="`printf '\n\t'`"

for file in `find . -depth` ; do
  [ "." = "$file" ] && continue                     # Skip "." entry.
  dir=`dirname "$file"`
  base=`basename "$file"`
  oldname="$dir/$base"
  newbase=`printf "%s" "$base" | tr A-Z a-z`
  newname="$dir/$newbase"
  if [ "$oldname" != "$newname" ] ; then
    mv -i "$file" "$newname"
  fi
done
```

11. Windows has serious filename problems too

Do *not* assume that filename issues are limited to Unix/POSIX/Linux systems; that is simply the focus of this particular paper. Windows also has serious filenaming issues, which in some ways are more serious than Unix/POSIX/Linux.

Windows forbids control characters in filenames, so it doesn't have that problem, and it forces an encoding, so they can be displayed unambiguously. But that isn't the only problem.

However, Windows has very arbitrary interpretations of filenames, which can make it dangerous. In particular, it interprets certain filename sequences specially. For example, if there is a directory called "c:\temp", and you run the following command from Windows' "cmd":

```
mkdir c:\temp
echo hi > c:\temp\Com1.txt
```

You might think that this sequence creates a file named "c:\temp\Com1.txt". You would be wrong; it doesn't create a file at all. Instead, this writes the text to the serial port. In fact, there are a vast number of special filenames, and even extensions don't help. Since filenames are often generated from attacker data, this can be a real problem. I've confirmed this example with Windows XP, but I believe it's true for many versions of Windows.

One solution is to prefix filenames with "\\?\" and then the full pathname; few people will do that consistently, leading to disaster. Web applications can protect themselves by only using filenames based on hashes, or forcing a prefix that makes the filename not a device name. (I have not been able to authoritatively confirm that only the usual lists of special names can be special, which makes this worrisome.) But this shows that Windows has its own serious filename issues.

The lesson here is not for POSIX to copy Windows; that would be a mistake. Instead, the goal is to have simple rules that make it easy to *avoid* common mistakes. Developers need systems that are neither "everything is permissible" nor "capricious, hard-to-follow rules that don't help users".

12. Some other opinions

I've received some interesting commentary on this article, both via email and via comments about it at lwn.net. My thanks to all the commenters. Not everyone agrees with this essay (I expected that!), but many did. Below are some comments that I found particularly interesting.

12.1. Ed Avis (epa on LWN.net)

Ed Avis said via email "*Hi, I read your fixing filenames essay - great work! I hope this longstanding problem is finally sorted out.*" He also suggested that "*A patch to lkml would at least get discussion moving, even if it has no chance of being accepted*".

On LWN, epa said:

> *I thoroughly agree. If using a single character for end-of-line was the best design decision in UNIX, then allowing any character sequence in filenames (while at the same time including a shell and scripting environment that's easily tripped up by them) was the worst.*
>
> *Look at the recent Python version that got tripped up by filenames that are not valid UTF-8. Currently on a Unix-like system you cannot assume anything more about filenames than that they're a string of bytes. This frustrates efforts to treat them as Unicode strings and cleanly allow international characters.*
>
> *Or look at the whole succession of security holes in shell scripts and even other languages caused by control characters in filenames. My particular favorite is the way many innocuous-looking perl programs (containing 'while (<>)') can be induced to overwrite random files by making filenames beginning '>'.*

12.2. Richard Neill

Richard Neill said via email:

1. *You might want to note that “,” is used by RCS to denote a versioned file, such as hello.txt,v Also, for ClusterNFS, the ‘$’ sign is used to denote that a file is specific to a certain client: eg /etc/hosts$$CLIENT$$ is substituted for the regular /etc/hosts when /etc is NFS exported.*
2. *Spaces in filenames are evil. But how do we get ordinary users to deal with them? What about taking the XMMS-style approach of displaying underscore as space.*

   So, add options to the GUI filemanagers (konqueror, nautilus etc) to: - display filenames with underscores as if they had spaces: “Hello_World” => “Hello World”. This is especially useful as it makes line-wrapping of filenames under icons look prettier. when the user types a filename with spaces, convert to underscore.

I agree that spaces in filenames are evil. But I suspect that we won’t be able to stamp them out widely enough to matter, because there are too many systems that absolutely require them. The Windows XP equivalent of “/home” is “\Documents and Settings” (notice the spaces!), and Windows’ equivalent of “/usr/bin” is “\Program Files” — so if you *ever* have to deal with Windows filesystems, trouble handling the space character is a real problem. (Vista and later use of “\Users” instead of “\Documents and Settings”, which is a more sensible Unix approach, but the problem still remains.) People do use both underscore and space in full pathnames, so making them effectively the same probably won’t work. If we can stamp out other problems, spaces in filenames become *much* easier to deal with, and I can live with that. That’s particularly true if we can get people to move to using an “IFS” setting without the space character — then they’re really easy to handle.

12.3. jreiser

On LWN, jreiser said:

Keep those filename rules out of my filesystems, please. Some of my programs use such “bad” filenames systematically on purpose, and achieve strictly greater utility and efficiency than would be possible without them.

But while jreiser may get “greater utility and efficiency”, lots of other people have programs that subtly fail possibly with security vulnerabilities, because of this leniency. I’d rather have “slower and working” than “faster but not working”. Such programs aren’t portable anyway; not all filesystems permit such names, and the POSIX standard doesn’t guarantee them either.

Interestingly, epa replies with:

Can you give an example [where ‘bad’ filenames are needed]? There is a certain old-school appeal in just being able to use the filesystem as a key-value store with no restrictions on what bytes can appear in the key. But it’s spoiled a bit by the prohibition of NUL and / characters, and trivially you can adapt such code to base64-encode the key into a sanitized filename. It may look a bit uglier, but if only application-specific programs and the OS access the files anyway, that does not matter.

12.4. nix

On LWN, nix was not so sure about this approach, and said:

I use the filename as a key-value store for a system (not yet released) which implements an object model of sorts in the shell... I pondered a \n-prepended filename because it’s even harder to trip over by mistake, but decided that it would look too odd in directory

> *listings of object directories when debugging... David's proposed constraints on filenames are constraints which can never be imposed by default, at the very least...*

The first part proves my point. Even for a key-value store, nix decided to avoid \n filenames because they cause trouble. If they cause trouble, then let's stop. I actually agree that some of these constraints cannot be imposed by default, but some can — so let's deal with those.

> *But I agree that these rules all make sense for parts of the filesystem that users might manipulate with arbitrary programs, as opposed to those that are using part of the FS as a single-program-managed datastore. What I think we need is an analogue of pathconf() (setpathconf()?) combined with extra fs metadata, such that constraints of this sort can be imposed and relaxed for *specific directories* (inherited by subdirectories on mkdir(), presumably). *That* might stand a chance of not breaking the entire world in the name of fixing it.*

This might the kernel (ha!) of a good idea. In fact, there's already a mechanism in the Linux kernel that might do this job already: getfattr(1)/setfattr(1). You could set extended attributes on directories, which would control what kinds of filenames could be created inside them. If this approach were implemented this way, I'd suggest that by default, directories would "prevent bad filenames" (e.g., control chars and leading "-"). You could then use "setfattr" on directories to permit badness, or perhaps enforce additional requirements. I would make those "trusted extended attributes" — you'd have to be CAP_SYS_ADMIN (typically superuser) to be able to make directories that permitted bad filenames. I'd like new directories to inherit the attributes of their parent directory, too; I'll need to look into that. I'm sure there are many other variations; much would depend on the viewpoint of kernel writers. This might give people the flexibility they want: Those who want reasonable filename limits can get them without rewriting the kernel, and those who want weird names can get them too.

12.5. ajb

On LWN, ajb said:

> *I think this is a sensible idea. It should be possible to make the transition relatively painless:*
>
> - *add a new inheritable process capability, 'BADFILENAMES', without which processes can't see or create files with bad names.*
> - *add a command 'access_bad_filenames' which creates a shell with the capability.*
> - */bin/ls also needs the capability, but should not display bad filenames unless an additional option is passed.*
>
> *That way, most processes can run happily in the ignorance of any bad filenames. If you need to access one, you run the commands you need to access it with under the special shell.*

I'm not sure about using capabilities this way, but it's certainly an interesting approach.

The basic notion of making this inheritable to processes is interesting. In fact, you could do inheritable shrouding of bad filenames solely from userspace, without the kernel. Simply define a special environment variable (e.g., HIDE_BAD_FILENAMES), and then modify programs so that they aren't found by programs that walk directories. You could probably just modify readdir(3)'s implementation, since I suspect other C routines, shells, and find(1) simply call that when they look for filenames. If not, I suspect the number of routines that need to be changed would be remarkably small. One trouble is that this might be *too* good at hiding bad filenames; you might not realize they exist, even when you need to find them, and attackers might intentionally create "hidden" files (e.g., so they can hide malware). Also, invoking setuid programs would erase this environment variable, and privileged programs are sometimes the programs you most want to *protect* from bad filenames. Which makes me worry; it'd be better to not have bad filenames in the first place.

You could also try to prevent *creating* such bad filenames from userspace, but here it gets dodgy. I suspect many programs invoke the kernel open() interface directly, and thus aren't quite as easy to intercept. And if we can't keep them from existing, they'll keep popping up as problems.

12.6. mrshiny

On LWN, mrshiny said:

> *You can pry my spaces from my filenames out of my cold dead fingers. But frankly spaces are no different than other shell meta-characters. If a filename is properly handled for spaces, doesn't it automatically work for all the other chars? If not, it should be easy enough to fix the SHELLS in this case.*

Well, spaces are actually different than other meta-characters in shells. The problem is that the default IFS value includes space, as well as tab and newline. As I discuss in the article, you CAN change IFS to remove space. If you do that, and ensure that filenames can't include newline or tab, then a lot of common shell script patterns actually become correct.

I agree with you, it's too late to forbid spaces-in-filenames on most systems. I thought I made that clear, sorry if I didn't. My point was that since most of us are probably stuck with them, let's get rid of some of the other junk like control chars in filenames; without them, spaces would be way easier to deal with.

> *Mr. Wheeler makes a mistake in the article as well. Windows has no problem with files starting with a dot. It's only Explorer and a handful of other tools that have problems. Otherwise Cygwin would be pretty annoying to use.*

You're right, the Windows kernel has no trouble with filenames beginning with dot. I was quoting something else, and didn't quite quote it correctly. Fixed. It's worth noting that to a lot of users, if the file Explorer has trouble, they have trouble. I'm an avid fan of Cygwin, BTW.

> *Overall, however, I like the idea of restricting certain things, especially the character encoding. The sooner the other encodings can die, the sooner I can be happy.*

Glad you liked the rest!

12.7. njs

On LWN, njs said:

> *I pretty much agree with all dwheeler's points (not sure about banning shell metacharacters).*
>
> *The section on Unicode-in-the-filesystem seemed quite incomplete. We know this can work, since the most widely used Unix *already* does it. OS X basically extends POSIX to say "all those char * pathnames you give me, those are UTF-8". However, there are a lot of complexities not mentioned here -- you need to worry about Unicode normalization (whether or not to allow different files to have names containing the same characters but with different bytestring representations), if there is any normalization then you need a*

> *new API to say "hey filesystem, what did you actually call that file I just opened?" (OS X has this, but it's very well hidden), and so on.*
>
> *But these problems all exist now, they're just overshadowed by the terrible awful even worse problems caused by filenames all being in random unguessable charsets.*

I did note that most people wouldn't be able to ban metacharacters.

Yes, I know about the issues with normalization. But my point is what you just noted — these problems are "overshadowed by the terrible awful even worse problems caused by filenames all being in random unguessable charsets". If you *know* that filenames will be handed to you in UTF-8 and won't have nasty stuff like control characters, many problems either go away or become more manageable.

12.8. kenjennings

On LWN, kenjennings said:

> *If you had a petition I'd sign it. I agree with all six of your fixes at the end of your article.*
>
> *Having been working with computers since 1979 and subject to the various limitations of dozens of file systems, I automatically exercise self-restraint and never put any of those characters into filenames.*
>
> *People should not be using filenames as data storage.*

12.9. Derek Martin

On Apr 15, 2010, Derek Martin sent me a lengthy and interesting email; here are some highlights (it's really long, so I don't include all of it):

> *I came across your article regarding Unix filenames. I mostly agree with a lot of your points, including that spaces in filenames are bad... As you point out, that's a hard one to get around, because spaces are allowed on a lot of other filesystems, and interoperation should be a goal of any system (ideally).*
>
> *I avoid using spaces in filenames, just as I avoid using control and meta characters in them. But I want to point out a couple of other issues, playing devil's advocate for a bit. The basic premise is, there's actually nothing wrong with the Unix filesystem allowing arbitrary character strings in filenames; the real problem is in the shell, and maybe a few of the standard command-line utilities.*
>
> *I'll start by pointing out that in part, Unix is where it is for the same reason Windows still honors a lot of MS-DOS brain damage: it simply was always that way...*
>
> *In the bad-old-days, there was no such thing as Unicode... [people could] specify their own language/encoding via environment variables, and [the kernel allowed] any sequence of bytes in filenames. This way, the implementers of the kernel don't need to be familiar with every character set in use in every language and culture...*
>
> *With Unicode, we don't really need to continue this practice. However, interestingly, using UTF-8 is not a complete solution to this problem, either... there are a few rarely used (but still used!) Korean syllabic characters, a number of Japanese-only characters (mostly typographical/graphical in nature), and a selection of uncommon Chinese characters that are not available in UTF-8, which are available in one or more of those*

languages' native encodings... UTF-8 contains enough of those languages' characters that any native user won't have trouble communicating; but some well-educated people may find their expressivity hampered.

Also, I must point out that most of the problems you've sited [sic] in your article are specific to the Unix shell... They are not problems inherent to Unix as a whole. Most other programming languages (C for example) have no trouble handling file names with odd characters in them. By and large, it just works (though displaying them or manipulating them in certain ways may still be an issue, if you can't identify their encoding). And where the use of GUI shells is now becoming common, even on Unix and Linux, this fact reduces the severity of some of the issues you outline. The GUI shells can handle those files just fine, for the most part. But back to the (Bourne-like) Unix shell, since that's what your article focuses on.

It should be (and is) possible to make a number of enhancements to the shell to allow better handling of such odd filenames. For example, something like the following could/should be possible:

```
$ var="\006\007xyz"
$ echo $var
\006\007xyz
$ echo "$var"
^F^Gxyz
```

One improvement: If unquoted, the shell could treat a variable containing spaces and control characters just as a C program would: i.e. they're not special....

My last point is filenames that start with a '-' character. That one is a little trickier, since a lot of tools don't have a way to handle it. There are tricks to do it... like specifying './-n' instead of just '-n' in your command. But, it must be pointed out that the magic '--' argument, while not implemented everywhere, IS defined in the POSIX standard. This is probably the best solution; sadly not everyone who writes programs is aware of and/or pays attention to standards. You can't blame that on Unix.

So, as a practical matter, since we don't currently have any of these things, I do still agree with you, mostly. But from a technical standpoint, the problems you outline are, I think, much more caused by the shell's poor handling of these special cases, than by the fact that they're allowed in the first place.

As far as Unicode/UTF-8 goes, Derek Martin is right, there is the problem that some very rarely-used characters aren't encoded in Unicode (and thus have no UTF-8 value). But that is almost never a significant problem, and this problem is slowly going away while these extremely rare characters are added to Unicode. More importantly, the world is different now. Today, people *do* exchange data across many locales, and it is simply unreasonable to expect that people can stay isolated in their local locales. Most people expect to be able to display filenames at any time, even though they receive data from around the world. We need a single standard for all characters, worldwide, and a standard encoding for them in filenames. There is really only one answer, so let's start moving there.

Martin notes that handling filenames beginning with "-" is tricky. Martin points out that the "magic '--' argument, while not implemented everywhere, IS defined in the POSIX standard. This is probably the best solution; sadly not everyone who writes programs is aware of and/or pays attention to standards. You can't blame that on Unix". Actually, yes, I *can* blame the standard. If a standard is too hard difficult to follow, maybe the problem *is* the standard. More importantly, even if programs implemented "--" everywhere, *users* would typically fail to use it everywhere. This is just like putting barbed wire on a tool handle; if a tool is difficult to use safely and correctly, perhaps the tool needs to be fixed. Anyway, the formal POSIX standard specifically states that you do *not* need to support filenames beginning with "-"; the problem is that many implementations permit them anyway. So we don't need to fix the standard; we just need to fix implementations in a way that complies with standards.

Martin says, "If unquoted, the shell could treat a variable containing spaces and control characters just as a C program would: i.e. they're not special...." Setting the IFS variable in the shell *does* make it possible to make the space, tab, and/or newline character nonspecial, so you don't even need to rewrite shells. I specifically recommend removing the space character from IFS, and that helps. That doesn't deal with the other characters, though.

Martin concludes, "So, as a practical matter, since we don't currently have any of these things, I do still agree with you, mostly." So we may disagree a little on their causes, but he still mostly agrees that something should be done.

Derek Martin claims that "most of the problems you've [cited] in your article are specific to the Unix shell...". I do talk about the problems specific to the shell, but the biggest problems with filenames are *not* specific to the shell or to command-line interfaces. The biggest problems are control characters, leading dashes, and non-UTF-8 encoding. Control characters are a problem for *all* languages, because essentially *all* programming languages have constructs that process lines at a time and handle tab-separated fields; control characters ruin that. Leading dashes interfere with invoking other programs, which is something that programs in *any* language sometimes need to do. The lack of a standard filename encoding means you can't reasonably display filenames, regardless of programming language or user interface. Certainly a number of other problems are unique to the shell, but that doesn't make them non-issues; the shell is so baked into the system, and used so widely (including via other programming languages), that they cause endless problems (including security problems). So let's fix them.

13. What to do?

Few people really believe that filenames should have this junk, and you can prove that just by observing their actions... Their programs... are littered with assumptions that filenames are "reasonable"... By changing a few lines of kernel code, millions of lines of existing code will work correctly in all cases, and many vulnerabilities will evaporate.

In sum: It'd be far better if filenames were more limited so that they would be safer and easier to use. This would eliminate a whole class of errors and vulnerabilities in programs that "look correct" but subtly fail when unusual filenames are created (possibly by attackers). The problems of filenames in Unix/Linux/POSIX are particularly jarring in part because there are so many other things in POSIX systems that *are* well-designed. In contrast, Microsoft Windows has a legion of design problems, often caused by its legacy, that will probably be harder to fix over time. These include its irregular filesystem rules that are also a problem yet will be harder to fix (so that "c:\stuff\com1.txt" refers to the COM1 serial port, not to a file), its distinction between binary and text files *, its monolithic design, and the Windows registry. Any real-world system has *some* problems, but the POSIX/Linux filename issues *can* be fixed without major costs. The main reason that things are the way they are is because "we've always done it that way", and that is not a compelling argument when there are so many easily-demonstrated problems. So let's fix the problem!

In general, kernels should emphasize mechanism not policy. The problem is that currently there's no mechanism for *enforcing* any policy. Yet it's often easy for someone to create filenames that trigger file-processing errors in others' programs (including system programs), leading to foul-ups and exploits. Let administrators determine policies like which bytes must never occur in filenames, which bytes must not be prefixes, which bytes must not be suffixes, and whether or not to enforce UTF-8. All that's needed in the kernel is a mechanism to enforce such a policy. After all, the problem is so bad that there are programs like detox and Glindra to fix bad filenames.

So what steps could be taken to clean this up slowly, over time, without causing undue burdens to anyone? **Here are some ideas:**

1. **Get standards bodies to clearly document the error code to be returned if a "bad" filename is rejected.** The reality is that Unix-like systems *already* mount filesystems with more restrictive naming rules (like MSDOS and NTFS), and must return errors, so it'd be best if they agreed on what to do in such cases. If possible, include in the standards some hints at likely conditions when a filename would be rejected, so that developers can avoid creating such (non-portable) filenames. For example, it could note that this is especially likely to occur if the filename includes control characters (bytes 0x00-0x1f), a leading dash in a component, punctuation not in the "portable" list, or is not a valid UTF-8 encoding. I suggest that EINVAL be the official error code that's returned when bad filenames are rejected; there is precedence for this, since the Linux msdos module already returns EINVAL when someone attempts to create a bad filename. James K. Lowden wrote to me and suggested that we need a new standard error code, EBADNAME. That's fine, too; I don't care *what* is agreed on, but there needs to be *some* agreement.

2. **Make it possible for system administrators to *easily* tell the kernel to reject creation of "bad" filenames (including via rename), and to decide if the system should hide files if their names are already bad. In short, implement filename enforcement in the kernel (the mechanism), and then let administrators decide what is "bad" (the policy)**. This could be implemented via an optional kernel module or optionally-enabled kernel capability. This capability could check filenames before they're created, and reject "illegal" filenames (perhaps returning EINVAL). Such a module could implement UTF-8 checking (which can be enabled or not) and 3 simple lists controllable by the system administrator: bytes forbidden everywhere, bytes forbidden as an initial character, and bytes forbidden as a trailing character. (Note that if a file name component is one character long, that one character is both the first and last character.)

   Merely forbidding their creation might be enough for a lot of purposes. On many systems, files are only created via the local operating system, and not by mounting local or remotely-controlled filesystems. On the other hand, if you also hide any such filenames that *do* exist, you have a complete solution — applications on that system can then trust that such "bad" filenames do not exist, and thus hiding such files essentially treats bad filenames like data corruption. I think that if you hide files with "bad" filenames, then you should reject *all* requests to open a bad filename... whether you're creating it or not. (One risk of hiding is that this creates an opportunity for malicious users to "hide" data in bad filenames, such as malware or data that isn't supposed to be there). Administrators could decide if they want to hide bad filenames or not, so there would be enforcement settings. Here is one possible scheme: One setting would determine whether or not to *permit* creation of files with bad filenames. Another would determine how they should be viewed if they are already there (e.g., in directories): as-is, hidden (not viewed at all), or escaped (see the next point)? Another would determine if they can be opened if the bad filename is used to open it (yes or no); obviously this would only have effect if bad filenames had been created in the first place. There would also be the issue of escaped filenames; if there is a fixed escaping mechanism, you configure which file wins if the the escaped name equals the name of another file.

   If bad filenames cannot be viewed (because they are escaped or hidden), then you have a complete solution. That is, at that point, all application programs that assumed that filenames are reasonable will suddenly work correctly in all cases. At least on that system, bad filenames can no longer cause mysterious problems and bugs.

   Let's talk about how this could be implemented in Linux, specifically. It could be a small capability built into the kernel itself. Josh Stone shows how filesystem rules could be implemented using SystemTap. In theory, it could be a pluggable Linux Security Module, but I suspect that won't work. People typically already have a big LSM module installed, there's more than one used by different distros, and there's no support for "stacking" multiple LSM modules in the stock kernel (stacking is a capability I helped develop early on, but it hasn't been accepted). It could be a separate pluggable module that's not an LSM module, using its own hooks. Another option is by creating a special pass-through filesystem, but the additional complexity such a filesystem would add doesn't seem necessary.

James K. Lowden informs me that "enforcement could be effected on NetBSD using a layered filesystem. [It] Would make a nice [Summer of Code] SoC project, too, as a proof of concept."

3. **Make it possible for system administrators to force "bad" filenames to be automatically renamed when returned to userspace**. Another obvious approach is rename bad filenames. This could be combined with the previous approach, e.g., forbid creating *new* bad filenames, but rename *existing* bad filenames (e.g., for filenames on an external network drive or a memory stick).

   There are many possible designs for a renaming system; here's a sample one:

   1. When the kernel sends a pathname to userspace, every byte that should be translated when going to userspace (as determined by the sysadmin) is converted to "=" followed by two hexadecimal uppercase digits. An "=" followed by two hexadecimal uppercase digits is always translated to "=3D" (an "=" that isn't followed by two hexadecimal digits is unchanged). If UTF-8-only mode is enabled, all non-UTF-8 bytes are translated (so userspace will only get UTF-8 sequences).
   2. When the kernel receives a pathname from userspace, every "=" followed by two hexadecimal uppercase digits that should be translated (as determined by the sysadmin) is translated into an internal one-byte code. Non-UTF-8 filenames are simply accepted.
   3. The sysadmin can set what is translated, by identifying what bytes are translated in either direction if they are the first byte, the last byte, or any byte in a filename. The admin must not include 0x00 and 0x2F ("/") and would usually would not include 0x2E ("."). A reasonable initial setup might be to set the "anytime" bytes to 0x01..0x1F and 0x7F..0xFF, the "begin" bytes to 0x2D ("-"), and the "end" bytes to 0x20 (" ").

   Let's examine various options; it turns out that there are many options to this, making it a little complicated.

   A common approach would implement an escape character (or escape sequence) that is used when the underlying filename is bad. This would also be a complete solution — users and developers could then truly trust that "bad" filenames can't happen (directory lists and so on would not produce them). The administrator could configure the specific policy of what filenames are "bad" for their system, using the same approaches described above (e.g., bytes forbidden everywhere, bytes forbidden as an initial character, bytes forbidden as a trailing character, bytes to be renamed everywhere/initially/trailing, as well as whether or not to enforce UTF-8).

   I presume that a file is stored in its "bad" form (if it's bad), is escaped (renamed) before being returned to userspace, and that any filename from userspace with the escape mechanism is automatically renamed back to the "bad" form when it is stored. The encoding character/sequence should itself be encoded, so that you do not have to worry about having two different files with the same user-visible name. This kind of "rename on create" isn't what most POSIX systems do, but MacOS already does this in some cases (it normalizes filenames with non-ASCII characters), and most application programs don't seem to care.

You're probably better off minimizing the number of filenames that will be renamed into a different sequence of bytes internally; this has implications on the encoding. For example, some encoding systems double the encoding character to encode the encoding character (so if "=" starts an encoding, then "==" can encode an "="). I had earlier suggested using doubling to encode the encoding character, but this violates the rule of minizing the renaming. In particular, if you use "=" in a filename at all, using "==" isn't unlikely (e.g., filenames like "==Attention=="). Unix/Linux filenames tend to have mostly or all lower case letters, so mandating that hexadecimal digits only be recognized if they are uppercase can help reduce unintentional renames too. To reduce the likelihood of unintentional encoding, I suggest having the kernel accept filenames and convert *only* filename components which have the encoding character followed by two hexadecimal digits, and where they are letters they must be upper case. Otherwise, any userspace "encoding" is not translated when brought to kernelspace, and conversely, an encoding from kernelspace is only used when it is necessary. Thus, if "=" starts an encoding and is followed by two uppercase hex digits, it would be encoded as "=3D", but filenames like "==Attention==" would not need to be renamed at all.

One of the problems with renaming systems is that many programs won't be prepared for low-level encoding and/or might permit filenames that can cause trouble later. At the very least, the kernel should not accept encodings for byte 0x00 nor byte 0x2F ("/"). It might be a good idea to forbid encoding byte 0x2E ("."). You only accept encodings that are currently forbidden, but that could would make it hard to change the rules later. Perhaps there should be a list of bytes which are translated from userspace, and all other "encodings" are ignored.

The same filename could appear different among different systems if it could be viewed at different times with and without encoding (e.g., perhaps it is stored on a memory stick, with the filename stored inside a separate file). This problem could be mostly alleviated by allowing programs to open or create files using unencoded bad names (including names that have encoding errors), while returning the encoded names when file lists are created later.

The administrator *could* also decide if the system allowed 'bad' filenames to be created (effectively renaming them on creation, from the point of view of user applications), or forbid their creation. This meant you could see existing data, but not create new problems. The rules could even be different (e.g., some "bad" filenames are so bad that they may not be created... others are okay, but will be escaped when viewed in a directory). But that becomes rather complicated. If useful, two simple settings could be added: should "bad" filenames be acceptable when creating files, and should "bad" filenames be acceptable when opening existing files. These settings might not be necessary, though; once renaming is automatic, bad filenames cannot cause that system any problem.

Finding a "good" escape character / escape sequence and notation is tricky; there seem to be problems with all characters and notations.

The "=" character is a particularly reasonable escape character; it is relatively uncommon in filenames, most programs don't consider a leading "=" as starting options, and it doesn't have special interactions with the shell. There's a lot of experience with this kind of thing; the quoted-printable format uses "=" as the escape character too. Basically, any "=" is then followed by two hexadecimal digits (uppercase for letters) which indicate the replaced byte value. You could encode the "=" sign itself as "==" or "=3D" or both; I suggest using "==" (doubling the =) as the preferred way to escape it, since that would be easier to read when someone *did* use an "=" in a filename. Then "foo\nbar" would become "foo=0Abar". This could also be used to escape names that aren't valid UTF-8 names. It could even be used to escape metacharacters and spaces, though I don't think everyone would want that :-).

An alternative would be "%" as the escape character, again followed by by "%" if the original character was "%", and a 2-digit hex value if it was any other forbidden character value. Then "foo\nbar" would become "foo%0Abar". One problem: using % is also the convention for URLs, and since URLs are often mapped directly to filenames, there might be interference. I think I prefer "=" over "%".

The "+" character is reasonable, though a few programs do use "+" as an option flag, and the built-in directory lost+found of filesystems would be renamed (making it slightly less good). One advantage to using "+" is that you could then use UTF-7 to encode the characters that need to be escaped; UTF-7 is at least widely implemented.

Some escape characters are especially bad. I suggest not using "\" (this is an escape character for C, Python, and shell) or "&" (an escape character for HTML/XML), because combining them could be very confusing. Avoid the main glob characters ("*", "?", and "[") — that way, accidentally omitting shell quotes is less likely to be painful.

An alternative would be to use a rarely-used UTF-8 character as the escape character; the escape character would take more bytes, and on some systems it might be harder to type, but that would reduce the "it's already being used" defense. Unfortunately, what is rarely-used for one person might be important to someone.

You could use an illegal UTF-8 prefix as the escape character, such as 0xFD, 0x81, or 0x90. This could be followed by two ASCII bytes that give the hexadecimal value of the bad byte. After all, if we have a bad character in the filename, it would be sensible to not produce a legal UTF-8 sequence at all. Then we can handle all legal UTF-8 sequences as filenames, since our escape mechanism can't be confused with legal UTF-8 sequences. The byte 0xFD is reasonable, since it is not legal in UTF-8 (it begins a 6-byte UTF-8 sequence, but more recent rulings such as RFC 3629 forbid 6-byte sequences). But using 0xFD more-or-less assumes that you will use UTF-8 filenames, since in many other encodings this may step on an existing character. The bytes 0x81 and 0x90 have some additional interesting properties: Not only are they illegal as a UTF-8 first byte, but these bytes are also not included in many Windows code pages (such as Windows-1252) and are not in many ISO/IEC code pages (such as ISO/IEC 8859-1). Thus, *many* people could use 0x81 or 0x90 as the escape sequence prefix to escape bad bytes in filenames (like control characters and leading dashes), even if they did *not* want to switch to UTF-8 filenames. Yet those using UTF-8 filenames could use exactly the same prefix. This means that we would not need to configure the prefix value, at least in many cases, and that makes many things easier. Between the two, I think I would pick 0x81, simply because it is the first value with the right properties. One negative of this is that this means that programs and programming languages will forever have to deal with illegal UTF-8 sequences in filenames, enshrining them instead of slowly getting rid of them.

You could also escape the bad byte as an overlong UTF-8 sequence, e.g., store the control characters 1-31 as two bytes instead of one. Then, if we receive a UTF-8 sequence that is *overlong*, we encode it back before storing it (while not allowing \0 and slash in the stored filename). One nice property of this is that display systems are more likely to display these correctly, if there is a way to display them at all (e.g., they may display leading dash as leading dash). Again, this more-or-less assumes you are using UTF-8 filenames for external (user-level) representation. A big problem is that some programming language libraries may read these overlong sequences in and convert them to ordinary Unicode characters; then, when they are written out, they could be written as ordinary (non-overlong) characters, changing their meaning. So, while at first I thought this made sense, now I think this is a bad idea.

A different approach would be to use an approach similar to Python PEP 383 encoding (though encoding non-slash bad bytes 1-127 as well). In short, encode each bad byte (other than ASCII NUL \0 and slash) to U+DCxx (the low-surrogate code points), then encode that with

UTF-8. This would include encoding bytes that are not valid UTF-8 in the underlying filesystem. The advantage of this approach is that PEP 383 encoding doesn't interfere with good filenames at all; it only renames bad filenames. Bad filenames would get a little longer (each bad byte becomes 3 bytes), but there shouldn't be many bad filenames in the first place, and many bad filenames only have a few bad bytes (unless they are due to encoding mismatch). Thus, newline \n (0x0A, aka U+000A) would become Unicode "character" U+DC0A, encoding to UTF-8 0xED 0xB0 0x8A. Similarly, a leading dash is ASCII 0x2D becomes Unicode U+DC2D, encoding to UTF-8 0xED 0xB0 0xAD. The largest possible bad byte is 0xFF, becoming Unicode U+DCFF, encoding to UTF-8 0xED 0xB3 0xBF. When the kernel gets a filename from userspace that includes UTF-8-encoded U+DCxx characters, they would be encoded back (except for encodings of \0 and "/", which would be ignored). If the filename stored on disk already has UTF-8 encoding of U+DCxx, it would be encoded again (so that when it is decoded later we end up with the original filename). Enabling this in some sense requires that filenames normally be UTF-8 (ASCII is a valid subset of UTF-8), since many other encodings would permit 0xED as a valid character, but it would work as an intermediate stage; if a filename uses a different encoding, it can still be found and then renamed to UTF-8. These might display in an ugly way, but that is often true even without encoding, and display systems *could* be taught to display these with "?" or some such. Such filenames are likely to be considered legal UTF-8, and thus programs that expect UTF-8 will like these filenames.

The specific escape sequence could be an administrator setting. Unfortunately, if it can be set, that will tend to make things *more* complicated, and we don't need *more* complications... it would be nicer to have a *fixed* escape sequence that we could count on.

One challenge is what to do about filenames that are so long that they "can't" be expanded; at that point, it may be better to simply not show the filename at all (and let specialized tools recover it). In practice, this is unlikely to be a problem.

My thanks to Adam Spragg, who convinced me to expand the description on doing renaming in the kernel.

4. **Modify kernels so that only specific processes and/or directories could create and view "bad" filenames; everyone else doesn't see them.** This might be implemented as a POSIX capability (e.g., perhaps you have to be CAP_SYS_ADMIN to create and see "bad" filenames). You could use Linux' setfattr(1) and getfattr(1) to create special directories where the rules are relaxed (and perhaps you have to have CAP_SYS_ADMIN to see in them).

   One problem with this approach is that programs that have extra privileges are *exactly* the programs that you *most* don't want fooled by misleading filenames. A sneaky alternative, which again could be a configuration option, might be that only privileged programs could *create* bad filenames, but only unprivileged programs could *see* them later. I think that alternative is amusing but a bad idea; better to forbid or escape bad filenames outright. In general, I think this is overly complicated, but I mention it for completeness.

5. **Try to fix things by intercepting low-level libraries**. You could set LD_PRELOAD so that important low-level C routines that work with filenames (such as open()) get intercepted as above. Unfortunately, this does not work with setuid/setgid programs, and these are the ones that most need protection. You could experiment using LD_PRELOAD, and then override using /etc/ld.so.preload, but that still does not handle statically-linked programs (SuSE ln for example). I don't think this is a good idea long-term, but it might be a very useful way to experiment to start with.

6. **Modify tools (both their POSIX specs and their implementations) so it's easier to ignore**

files with bad filenames. This would probably focus on the shell, find(1), and various C library routines (like readdir(3), readdir_r(3), and scandir(3)), since much of the initial work in generating file lists occurs with them. Here are a few ideas:

- Add standard tests to "find" so that applications can easily ask it to skip "bad" filenames. At the least, add "-skipacontrol", which would cause find to automatically skip any filename with an ASCII control character in it, that is, 0x00-0x1f and 0x7f. (There's no need or desire to make this locale-dependent; the fundamental problem is with the original ASCII set.) Other options might be "-skipdash" (skip filenames with a leading-dash component), "-utf8" (only provide filenames that are legal UTF-8), and even "-clean" (only provide filenames that pass -skipacontrol, -skipdash, and -utf8). Similarly, add "-hidden" so that "! -hidden" could be a more accurate replacement for globbing. These options would make it easier to write programs that skip badly-named files, which would eliminate a lot of bugs and security problems (because programs wouldn't accidentally process these bad names). It might also hasten the day when people agreed they weren't useful, since they were getting quietly filtered out by many programs.
- Add a standard ability for Bourne shells' globbing so that filenames with control characters and components beginning with "-" can be automatically skipped when globbing. It'd be nice if this were something accessible via *set*, so that it'd be easy to do as part of a "standard prefix". Globbing normally skips files beginning with "." already, so it's not like globbing gets all filenames anyway, and such filenames are more trouble than they're worth. Bash's nonstandard GLOBIGNORE extension lets you skip formats, but it's not standard, nor is it particularly easy to use for this purpose. (It'd also be nice to add support for bash's "nullglob", so that failed globbing simply returned an empty list (as it sensibly should), and bash's "dotglob", so that shell programs could easily include hidden files if they wished. There isn't a perfect solution for handling globbing when there are no matches and they aren't in the iteration range of a loop.)

7. **Modify tools (POSIX spec and implementation) so difficult filenames (including filenames with embedded blanks) are automatically handled.** For example:
   - *Leading dashes*: Whenever globbing or directory scanning is done, prepend "./" to any filename beginning with "-". In the case of globbing, you could do this prepending the pattern *could* match an initial dash. This could be implemented in glob(3), scandir(3), readdir(3), and any shells that implement globbing themselves (instead of calling glob(3)). This would mean that "cat *" would return "./-n " instead of "-n" if such a file existed. I suspect it'd be a lot more acceptable to add an option to glob(3), and then make it the default or talk shell implementations into using it, than to modify readdir(3) which is lower-level — but that does mean that programs using readdir directly would still be vulnerable. The POSIX spec says that filenames beginning with "-" aren't portable, so I think you could claim POSIX conformance even with this, on the grounds that permitting such filenames at all is a non-standard "extension". Note that merely fixing one Bourne shell implementation (such as bash) is not enough, because there are many shells in use. For example, Ubuntu uses *dash* as its /bin/sh shell yet interactive users use bash... while on Fedora many users use bash for both. Many embedded systems use the busybox shell. So whatever is done needs to work across many shells.
   - *Whitespace in filenames*: For bonus points, you could work for a future day where IFS is set to "\n\t" by default in system configurations (at least for interactive shells, and possibly as an inherited variable to all shells). I think that this is improbable; even if it happens, it will not happen in my lifetime :-(. But it's an idea.

8. **Modify tools (POSIX specs, implementations, and their documentation) so that it is *much easier* to write programs that correctly handle at least some bad filenames.** If you really disagree with my premise, and say that it is important for POSIX filenames to be able to include

arbitrary junk (er, bytes), and for programs to actually accept and process all such filenames correctly, then something must change. It's too hard to do it correctly today; witness all the not-quite-right programs out there, and the ugly approaches that are necessary to be "correct". If ordinary filesystem traversal is hard to do correctly, then "Joe Developer" (and even smart busy developers) will not do them correctly. Where possible, it's great to do it automatically (see the previous point), but that's often not possible. So what can we do to make it easier? Here are some thoughts:

- **Make it much easier to handle \0-separated streams.** This is, at least, already supported by a few tools (though with inconsistent option names). Pick a standard option name (-0 or -z, say) that can be used consistently everywhere, standardize it in POSIX, and get *lots* of tools to implement it, both shells and non-shells.

  Shells are especially important, because you want their command substitution, variable substitution, and "read" operations to be able to *easily* do splitting on only \0. You'd need to modify all the Bourne-like shells so that they could use \0 as a field splitting character, either by default, new syntax, or via some easy setting (e.g., a glob setting). The zsh shell can include \0 inside variables but many shells cannot. It would probably be easier to get wider acceptance if mechanisms that *all* shells could easily support were created, even if the shell does not support \0 (null byte aka NUL) inside variables. This is *especially* important for the `for` loop, as this is the easy way to loop over returned filenames. The idea would be to make something like `` IFS="" for x in `find . -print0` `` or some such to split cleanly on \0 (it doesn't today on most shells). No, `IFS=$'\0'` doesn't work in bash 3.2.39 (unsurprisingly; C programs often don't like multicharacter strings that contain \0). It really should be the default; if it's not, devising good syntax for this is tricky! One possibility is to specially interpret a 0-length IFS value in the `for` loop as splitting on \0. Another possibility would be to devise another special setting or syntax that meant "when splitting, ignore IFS and split on \0 instead". Then modify the "for" loop syntax to be `"for name [ [using word [ in word ] ] ; do list ; done"`; the "using" stuff could set the \0 setting or an IFS value, which would apply *only* to the `in word` part. Another variant would be to allow `null in` or `zero in` instead of `in`, meaning to split on \0 instead of using IFS. Hopefully someone will come up with something better! The Bourne-like shells' "read" command will also need to be able to easily read \0-delimited values; you already can do this in bash 3.2.39 using the -d option, e.g., `IFS="" read -r -d ''` ("don't use IFS to split this up at all, don't interpret backslash specially, and the delimiter is \0"). But not only is this nonstandard — don't you see how complicated that is? It has to be *really easy* to use, like "`read -0`", or people will forget to use it.

  It's not just shells; there are a lot of other tools that might need to generate or accept filename lists, and they'd all need to be modified to handle \0 as a separator. The programs "find", "xargs", and "sort" are obvious, but almost anything that does line-at-a-time processing might need to support \0 as a possible separator instead, and almost anything that might generate or use filename lists will need to be modified so it can use \0 as the separator instead of newline (or whatever it normally uses). And you can't change just one implementation of a command like "sort"; you'd have to modify GNU's sort and BSD's sort and busybox's sort and so on. Of course, after modifying all these infrastructure utilities, you'd have to modify every program that processes filenames (!) to actually use these new abilities. You could try to use an environment variable that means "default to using \0 separators", but turning on such a variable will probably mess up many programs that *do* work correctly, so I have little hope for that. And after all this, displaying filenames is still dangerous (due to terminal escapes) and inconsistent (due to a lack of standard encoding).

- **Or, create and implement a standard filename escaping mechanism at the tool level**. If you don't like the "\0 everywhere" approach, you could define some standard "escape" format for filenames. Then you have to make sure that filenames are escaped before being handed to programs, and then make sure people un-escape filenames just before use. There'd have to be some agreement on the convention; GNU's "find" and "ls" have a C-like escaping mechanism that's already widely available, so I think that one has the best chance. Now you need to modify lots of other programs to generate or accept this format (e.g., xargs). But now people have to insert escaping and un-escaping codes everywhere — better make sure that's *really* easy, and that it's widely supported!! And again, you now need to modify nearly every POSIX/Linux/Unix program on the planet to use these conventions, since they're created by applications instead of being implemented in the kernel or low-level library. Are you *really* going to get people around the world to do all that escaping and un-escaping? That isn't simple; that is a crazy amount of work. I think this approach is unlikely to work — too much pain, not enough benefit.
- **Create standard tools for encoding/decoding filenames**. See my encodef home page for more information on a toolpair to do this. One problem is that, without *some* extensions of other tools, these can be annoying to work with. So you still want some additions to standard tools (e.g., xargs and find should support "\0" encoding) so you have something to build from. Unfortunately, this implies that developers would have to use these encodings *everywhere* to do any good; I suspect many won't do it, or won't do it consistently.
- **Modify unpacking/remote-copy tools to automatically rename filenames.** Modify unpackaging tools (like tar, bunzip, and unzip) and remote-copy tools (like scp and rsync) so that they can automatically rename "bad" filenames into "good" ones. In particular, they need the ability to do perform character encoding conversions (they can just call "iconv(1)" to do all the work; they don't need to build in the capability themselves). While you're at it, may as well have them eliminate other nasties like leading dashes and control characters. They really should have this capability anyway; today, there are already some filesystems that forbid bad encoding. Combined with the lack of standardized encoding, users have to rename filenames anyway. Best to build this into the programs that (re)create the filenames.
- **Simplify shell "read" for easier space, tab, and backslash handling.** Add a standard option to the Bourne shell's "read" that automatically has the effect of `IFS="" read -r -d ""` so it'd be easier to read in null-terminated lines without corrupting the result. That does not completely solve the problem, for example, filenames with newlines and tabs will still not be handled correctly unless they cannot happen in the first place.
- **Add standard arguments to xargs**. The POSIX standard version of xargs is absurdly hard to use; modify the standard so that it's easy to use. In particular, xargs is nearly always used with "find", yet the input format of xargs is incompatible with the default output format of find. At the least, add support for the GNU -d (--delimiter) option so the simple one-filename-per-line format is easy to process, and add support for the GNU -0 (--null) option to support null character termination of input items. Both of these options should dissable the "end of file" string of -E, and disable the parsing of whitespace, quotes, and backslash. Then, get these implemented everywhere. The GNU, BSD, and even busybox implementation of xargs has -0, so that option in particular is already widely supported.
- **Add a "simplified quoting" mode to Bourne shells**. Often nearly all variable references must be surrounded by double-quotes, except in the few places where breaking up a variable is desired. This is a bad default. It'd be nice to add a "set" mode so that by default, variable references were interpreted as if they were already surrounded by double-quotes, and then add a special referencing mode or operator that split results. This is how most other languages handle variables. Perhaps this should be broadened to other substitutions, too.
- **Add support for `$'...'` in shells.** Modify the POSIX spec and implementations

(especially dash and busybox) so that all shells support the `$'...'` extension, making it easier and more efficient to set IFS. Then, get people to add `IFS=$'\n\t'` when writing shell scripts as much as they can (so constructs like ``for x in `find .` `` will work correctly on filenames containing spaces). This makes it slightly easier to handle filenames with spaces. There are other good reasons to do this (speed and convenience) that are beyond the scope of this paper, and I would like to see this done. I think `$'...'` will in the next version of the POSIX specification; you can blame me for proposing it. I think this is a nice improvement (for other reasons), but doing this does not really address most of the problems discussed in this paper.

Few people really believe that filenames should have this junk, and you can prove that just by observing their actions. Their programs, when you read them, are littered with assumptions that filenames are "reasonable". They assume that newlines and tabs aren't in filenames, that filenames don't start with "-", that you can meaningfully and safely print filenames, and so on. Actions speak louder than words — unless it is easy, people will not do it. Continuing to allow filenames to contain almost anything makes it very complicated to have correctly-working secure systems. I'm happy to help with the "make it easier" stuff, but in the long run, I don't think they're enough. By changing a few lines of kernel code, millions of lines of existing code will work correctly in all cases, and many vulnerabilities will evaporate.

This won't happen overnight; many programs will still have to handle "bad" filenames as this transition occurs. But we can start making bad filenames impossible now, so that future software developers won't have to deal with them.

What is "bad", though? Even if they aren't universal, it'd be useful to have a common list so that software developers could avoid creating "non-portable" filenames. Some restrictions are easier to convince people of than others; administrators of a locked-down system might be interested in a longer list of rules. **Here are possible rules, in order of importance** (I'd do the first two right away, the third as consensus can be achieved, and the later ones would probably only apply to individual systems):

1. **Forbid/escape ASCII control characters (bytes 1-31 and 127) in filenames, including newline, escape, and tab.** I know of no user or program that actually *requires* this capability. As far as I can tell, this capability exists only to make it hard to write correct software, to ease the job of attackers, and to create interoperability problems. Chuck it.
2. **Forbid/escape leading "-".** This way, you can always distinguish option flags from filenames, eliminating a host of stupid errors. Nobody in their right mind writes programs that *depend* on having dash-prefixed files on a Unix system. Even on Windows systems they're a bad idea, because many programs use "-" instead of "/" to identify options.
3. **Forbid/escape filenames that aren't a valid UTF-8 encoding.** This way, filenames can always be correctly displayed. Trying to use environment values like LC_ALL (or other LC_* values) or LANG is just a hack that often fails. This will take time, as people slowly transition and minor tool problems get fixed, but I believe that transition is already well underway.
4. **Forbid/escape leading/trailing space characters — at least trailing spaces.** Adjacent spaces are somewhat dodgy, too. These confuse users when they happen, with no utility. In particular, filenames that are *only* space characters are nothing but trouble. Some systems may want to go further and forbid space characters outright, but I doubt that'll be acceptable everywhere, and with the other approaches these are less necessary. As noted above, an interesting alternative would be quietly convert (in the API) all spaces into unbreakable spaces.
5. **Forbid/escape "problematic" characters that get specially interpreted by shells, other interpreters (such as perl), and HTML/XML.** This is less important, and I would expect this to happen (at most) on specific systems. With the steps above, a lot of programs and statements like "cat *" just work correctly. But funny characters cause troubles for shell scripts and perl, because they need to quote them when typing in commands.. and they often forget to do so.

They can also be a cause for trouble when they're passed down to other programs, especially if they run "exec" and so on. They're also helpful for web applications, again, because the characters that *should* be escapes are sometimes not escaped. A short list would be "*", "?", and "["; by eliminating those three characters and control characters from filenames, and removing the space character from IFS, you can process filenames in shells without quoting variable references — eliminating a common source of errors. Forbidding/escaping "<" and ">" would eliminate a source of nasty errors for perl programs, web applications, and anyone using HTML or XML. A more stringent list would be "`*?:[]"<>|(){}&'!\;`" (this is Glindra's "safe" list with ampersand, single-quote, bang, backslash, and semicolon added). This list is probably a little extreme, but let's try and see. As noted earlier, I'd need to go through a complete analysis of all characters for a final list; for security, you want to identify everything that is permissible, and disallow everything else, but its manifestation can be either way as long as you've considered all possible cases. But if this set can be determined locally, based on local requirements, there's less need to get complete agreement on a list.

6. **Forbid/escape leading "~" (tilde)**. Shells specially interpret such filenames. This is definitely low priority.

In particular, ensuring that filenames had no control characters, no leading dashes, and used UTF-8 encoding would make a lot of software development simpler. This is a long-term effort, but the journey of a thousand miles starts with the first step.

Feel free to see my home page at http://www.dwheeler.com. You may also want to look at my paper Why OSS/FS? Look at the Numbers! and my book on how to develop secure programs.