Chapter 12
Directory Traversal Attacks

Revision 3.2, November 2025.

Objectives

- Introduce basic file system concepts and terminology to understand the problem.
- Understand the directory traversal attack (or path traversal attack) with examples.
- Learn how to mitigate this type of attack.
- Learn the difference between allowed and blocked list approaches to limiting access.

12.1 Background

A *directory traversal attack* (sometimes called a *path traversal attack*) occurs when the program constructs a file path name using input from the user, such that it results in accessing an unintended file. In other words, there is a path from the attack surface to the point where the file path name is constructed and used. For this reason, we often refer to this type of vulnerability as being subject to a *path name injection* attack.

These attacks are similar to other injection attacks in that instead of a simple file name, the attacker enters special elements that change the meaning of the resultant path to reference other files never intended to be possible. In the case of file names, we are most interested when "`.`", "`..`", "`/`" (Unix), and "\" (Windows) appear in a file name. Any time that a program constructs a path name to access a file, it is imperative to prevent malicious inputs from manipulating the resultant path in unexpected ways.

12.2 File System Basics

First, we will review some file system basics to better understand the mechanisms that will be used to form these attacks. If you have a strong background in operating systems, you can skip this section and jump ahead to Section 12.3.

12.2.1 Hierarchical File Names Path Names

Our modern notion of a file system was invented as part of the Multics operating system research project at MIT[1] in the early 1960's. The idea of a

[1] `https://en.wikipedia.org/wiki/Multics`

hierarchical file system comes from a seemingly simple idea: a directory can contain data files *and* other directories. Previous to Multics, directories could only contain data files. If a directory can contain other directories, then the collection of files stored by the computer will form a tree, as shown in Figure 1. Another way to say this is that a file can be a normal file that contains data, or it can be a directory file that points to other files. To name a specific file, we start at the directory at the top of the tree, called the *root directory,* and describe the path through the file system tree to the file that we wish to access.

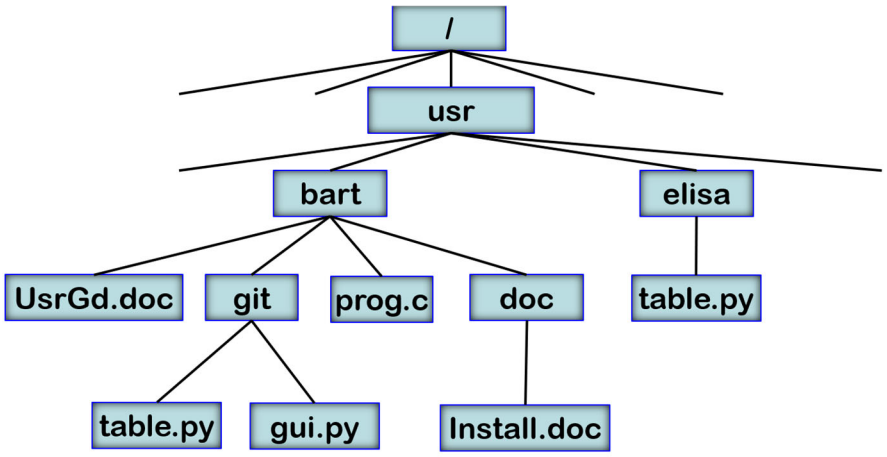

Figure 1: Fragment of the System File Tree

File path names consist of identifiers concatenated together with path separator characters to name a component of the hierarchical file structure. Unix paths use slash ("`/`"), and Windows paths use backslash ("`\`") as path separator.

Path names can also contain a couple of special elements. "`.`" means the current directory and "`..`" means the parent (containing) directory. There will be more details on these special elements as we go through this chapter. These special elements will play an important role in formulating directory traversal attacks.

12.2.2 Absolute Path Names

Absolute path names always start at the root of the file system. In Unix, absolute path names begin with a slash ("`/`") and in Windows, they begin with either "`\\`" or naming a drive by letter such as "`C:\`". Now we will look at some example absolute path names. Each of these examples refers to a file illustrated in the directory tree in Figure 1. As we go through each example, refer to the directory tree to see how the name traverses the tree.

Unix syntax	Windows syntax
`/usr`	`\\usr`
`/usr/elisa/table.py`	`\\usr\elisa/table.py`
`/usr/bart/doc/Install.docx`	`C:\usr\bart\doc\Install.docx`

The first example above names the directory `usr` in the root directory.

The second example names the file `table.py` contained in directory `elisa`, which is contained in directory `usr`, which is contained in the root directory.

The third example names the file `Install.docx` contained in directory `doc`, which is contained in directory `bart`, which is contained in the root directory (on Unix systems) or in drive C (on Windows).

12.2.3 Relative Path Names and Current Working Directories

Relative path names refer to file path names that start at the *current working directory (CWD).* For each running program, or process, there is a CWD. Any file name reference that does not begin with the root directory starts at the CWD.

When a process is created (i.e., when a program starts to run), the CWD is set to the CWD of its *parent* process, that is the process that created the new (*child*) process. So, if you start a program from the shell, the new program's CWD is the same as that you set for the shell.

When you initially log into a system, the CWD for your login shell is set to the directory assigned to hold your files, called your *home directory*. From the shell, you can change your home directory using one of these commands:

`SetCurrentDirectory` (Windows)
`cd` or `chdir` (Unix)

Now we will look at a couple of simple examples of relative path names. Again, these file names are contained in the directory tree in Figure 1. Follow along in that tree as the file names are described.

Unix syntax	Windows syntax
`gui.py`	`gui.py`
`git/gui.py`	`git\gui.py`

The first example above refers to the file `gui.py` in the CWD. Here, the CWD was set to `/usr/bart/git`, resulting in access to `/usr/bart/bit/gui.py`.

The second example refers to the file `gui.py`, which is contained in directory `git`, which is contained in the CWD. Here, the CWD was set to `/usr/bart`.

12.2.4 Path Names Using ".", "..", "~", and Multiple Slashes

When we introduce special elements into a path name, things get a bit trickier. These elements are quite useful in normal use of a computer but can also be used to try to avoid protection and safety checks on which files a user is allowed to access. We illustrate the use of these elements with the following examples. Once again, you can follow along in the directory tree show in Figure 1.

Unix syntax	**Windows syntax**
`/usr/elisa/../bart/git/gui.py`	`\\usr\elisa\..\bart\git\gui.py`
`../elisa/./table.py`	`..\elisa\.\table.py`
`~elisa/table.py`	`~elisa/table.py`

The first example above is based on an absolute path. Going through the path name left to right, we will describe how the file system is traversed to find the file. Starting at the root directory, we find file `usr`, which is a directory. In that directory, we find file `elisa`, which is also a directory. In that directory, we find file `..`, which moves us up one level to the parent directory `usr` again. From there, we find file `bart` (a directory), and in that directory find file `git` (another directory), and finally find the file `gui.py`.

The second example above is a relative path, assuming that the CWD is set to `/usr/bart`. The "`..`" refers to the parent directory, `/usr`, and the file `elisa` takes us into that directory. The dot ("`.`") keeps us in the same directory (`/usr/elisa`) and finally `table.py` names a file contained in that directory.

The third example uses the ~ (tilde) character, which refers to the user's home directory. So `~elisa` is a shorthand way of writing `/usr/elisa` in this example. The use of ~ simplifies finding a home directory, as home directories might be located in different places on different systems. On one system, it might be `/usr/elisa`, and on another system it might be `/home/elisa` or `/usr/home/elisa`.

One additional way to complicate path names without changing their meaning is to add multiple slashes in a row. For example, `/a/b/c` is equivalent to `//a///b/c`. Making path names more complicated is often a way to confuse code that is trying to restrict access to specific files.

12.2.5 Canonical Path Names

The canonical path name is the unique path name for a file or directory with all the special elements removed, including "`.`", "`..`", and "~"; as well as symbolic links resolved. For the first example in the table above, the

canonical path name of the file (in Unix syntax) is `/usr/bart/git/gui.py`. For the second example, the canonical path is `/usr/elisa/table.py`, assuming that the CWD was set to `/usr/bart`.

Most operating systems and programming languages provide built-in functions or methods to produce the absolute or canonical path name from a file name. The table in Figure 2 shows some of these functions and methods.

This description of path names only introduces the basics so that you can understand how path name attacks can happen. It is not intended to provide the complete details that might vary depending on the specific operating system. It is well worth studying the nuances and quirks of the platform on which you work in detail to understand potential additional attacks that may be possible.

System	Absolute Path	Canonical Path
Linux		`realpath` `canonicalize_file_name`
Windows		`PathCanonicalize`
Java	`getAbsolutePath`	`getCanonicalPath`
Python	`os.path.abspath`	`os.path.realpath`
Ruby	`File.absolute_path`	`Pathname.realpath`
Perl		`Cwd::abs_path` `Cwd::real_path`

Figure 2: Commonly Available Functions/Methods to Get Absolute or Canonical Path Names

12.3 Directory Traversal Attacks and Attempted Defenses

Now, we will discuss how file names can be used to form an attack. When software constructs file path names from user input without proper checks or sanitization, there is the opportunity for a malicious user to circumvent file access checks to read, write, or even delete files that they are not authorized to do so.

We start our discussion of attacks by looking at a couple of scenarios where user input is legitimately used to construct file path names. We then look at some ways that people have used to try to defend again directory traversal attacks and ways to attack that code.

12.3.1 A Couple of Directory Traversal Attack Scenarios

It is common structure for a server to have a directory in the file system that contains a subdirectory for the user accounts (here, we will call it `useraccounts`). In that `useraccounts` subdirectory, we have one

subdirectory for each valid user, where the name of the subdirectories are often based on the user ID. This structure is illustrated in Figure 3, where we have directories for accounts `admin`, `arnold`, `elisa`, and `bart`. In each of these user directories are the various files that need to be stored on a per-user basis.

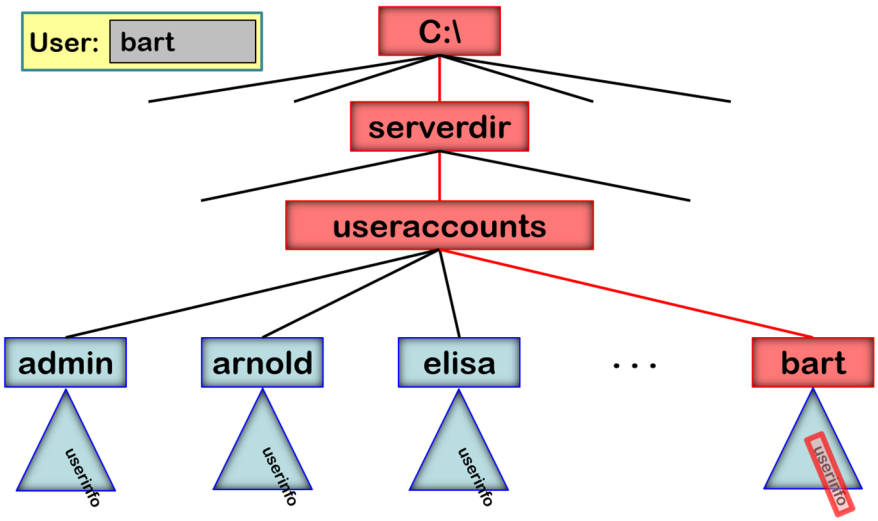

Figure 3: Using User Input to Identify Per-User Files

If the server wanted to access the `userinfo` file for the user `bart` who just logged on, it would construct the file name by concatenating three components together.

We start with the path name of the directory that stores the user accounts, which in this case is `C:\serverdir\useraccounts\`.

When user `bart` logs in, we append the user name onto the file name string, and then append the particular file name that we wanted to access in that directory. This gives us the full path to the file"

"`C:\serverdir\useraccounts\`" + "`bart`" + "`\userinfo`"

The most important part of this file pathname is the user ID, which came from the attack surface so can be manipulated by an attacker.

A directory traversal attack is based on the user crafting input to try to cause access to a file to which they should not have access.

Of course, we have used a Windows file name in this example, but this is equally applicable to Unix systems.

Our second scenario starts with the same structure, where the server maintains a directory per user where the user can keep files. For example, if the server was storing files that contained information about cargo shipping containers, there might be documents that describe the contents of the container, customs information, and hazardous materials information. In this scenario, the user wants to upload a file called `customs.pdf` to their account. This scenario is illustrated in Figure 4.

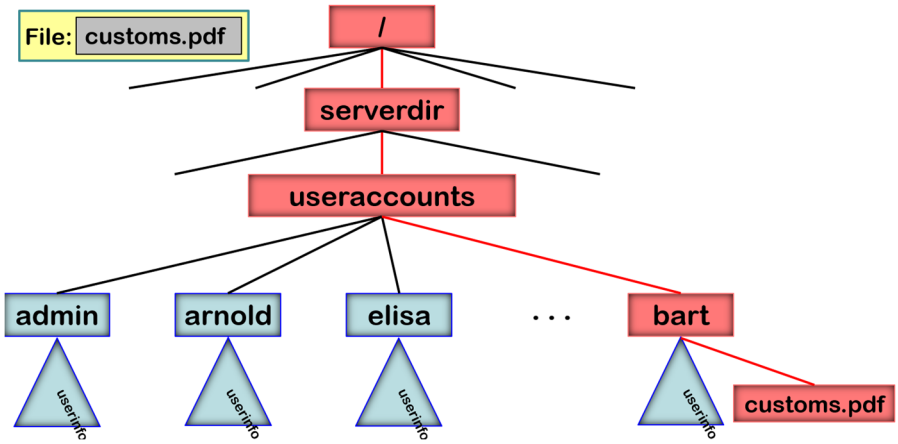

Figure 4: Using User Input to Identify a Specific File Name

To construct the file path name, we start with the directory formed from the base directory and user name and then append the name of the file that the user specified.

"`/serverdir/useraccounts/`" + "`bart`" + "`/`"+ "`customs.pdf`"

Since the file name comes from the attack surface, there is an opportunity for the attacker to manipulate the resulting path name.

12.3.2 Vulnerability in a Naïve Implementation

To see how these attacks work from the perspective of the code, we begin with a simple example where the code was written with no consideration of security threats. In this example, a web service takes a parameter named `file` that is expected to correspond to the name of a file already in the server's `/safedir` directory. The request to the server will cause the specified file to be deleted. The files residing in `/safedir` belong to the user, so such an operation should be normal and safe.

Note that since the `file` parameter comes from user input, it can be manipulated by the attacker.

```
01 String filename = request.getParameter("file");
02 pathname = "/safedir/" + filename;
03 File f = new File(pathname);
04 f.delete();
```

Figure 5: Code to Delete a File on the Server (Vulnerable)

Figure 5 is a simple (and vulnerable) version of the delete code in the server. It is invoked with a `file` parameter with the value "`../etc/passwd`". The first line sets the variable `filename` to that value.

Line 2 creates the path name string `pathname` with the value "`/safedir/../etc/passwd`".

Since the directory `/safedir` is in the same directory as the directory `/etc`, the effective path name produced on line 2 is `/etc/passwd`, with a result that the file containing the passwords for the system is deleted on line 4.

The programmers of this system were aware of this type of attack so attempted to add mitigation code, as shown in Figure 6. This version of the code is actually based on a vulnerability that we discovered in our software assessment work.

```
01 String filename = request.getParameter("file");
02 String pathname = "/safedir/" + filename;
03 // remove ../'s to prevent escaping out of /safedir
04 pathname = pathname.replace("../", "");
05 File f = new File(pathname);
06 f.delete();
```

Figure 6: Code to Delete a File on the Server (with First Attempt at Mitigation)

This code is invoked with the same `file` parameter with the value "`../etc/passwd`". And line 2 creates a `pathname` with the same value as before, "`/safedir/../etc/passwd`".

To defend against access to the parent directory, the `pathname` string is transformed on line 4 to remove the "`..`" in the hope of avoiding this problem. The result is that `pathname` now has the value "`/safedir/etc/passwd`". This new name now describes a file contained in the intended directory (if it actually exists). Access to the parent directory was prevented.

However, this mitigation is overly simplistic and restrictive. The clever attacker's next attempt provides a `file` parameter with the value "`..../etc/passwd`".

When line 2 creates the `pathname` string variable, it has a value of "`/safedir/....//etc/passwd`". Line 4 modifies `pathname` to have a value of "`/safedir/../etc/passwd`"

This path name breaks out of the intended directory, resulting in the path "`/etc/passwd`", which results in deletion of all passwords just the same as we saw in the unmitigated example. As this weakly mitigated example shows, ad hoc attempts to block these attacks are risky.

More importantly, the mitigation in Figure *6* is trying to fix the malicious input, which is a dangerous and unproductive approach.

In general, it is better to avoid trying fixing malicious input and instead check the safety of the input and reject any inputs that are trying to make inappropriate file accesses.

12.3.3 Another Subtle Kind of Attack

Here is another example from a real system that tried to prevent a directory traversal attack. The approach used in this code is attractive because it is checking the input and rejecting it if it is invalid. However, unfortunately, this code is vulnerable to an attack based on an unexpected behavior of the Java `File` class.

Here, the programmer decided to reject user input if the string contained a file path separator character, which is "`/`" on Unix systems (which includes Linux) and "`\`" on Windows. Since they were accepting only simple file names and not path names, this seems to be a simple and direct strategy. To make their code portable, they used the built-in definition, `java.io.File.separator`. This definition returns the right character for the operating system on which the program is running. We show the code implementing this check in Figure 7, with the key check on line 4.

```
01 String path = request.getParameter("file");
02 String fullpath = "C:\\safedir\\"+path;
03 // Check for dir separators to prevent escape from safedir
04 if (path.contains(java.io.File.separator)) {
05   throw new PathTraversalException(path+" invalid");
06 }
07 File f = new File(fullpath);
08 f.delete();
```

Figure 7: Code to Delete a File on the Server (Mitigation by Detecting Separator Characters)

There is an interesting and subtle behavior of Java that subverts this approach to checking the input. When Java runs on Windows, a file path name can

contain either "/" or "\" as valid separator characters. Java on Linux systems is less egalitarian; only "/" is accepted as a valid separator character.[2]

In the above code, we have a mismatch. On Windows, the `java.io.File.separator` is "\", however file path names can also use "/". So, if a malicious user provides the string

```
../Windows/System32/Boot/winload.exe
```

to the above code, the effective path name of the file that is deleted will be:

```
C:\Windows\System32\Boot\winload.exe
```

As a side note, a careful inspection of the code in Figure 7 shows that there is an additional problem when the input file name is only "`..`".

12.4 Directory Traversal Mitigations

There are three different approaches that you can take to prevent directory traversal attacks, depending on the type of file names that are excepted. The first and simplest approach is to use a *blocked list* to forbid inputs that have illegal characters in them. The second approach is to use an *allowed list* to accept only inputs that consist of valid characters. The third and most general approach is to use canonical paths to check that the files being accessed are within the allowed directory. We describe each of these approaches in the following sections.

12.4.1 Preventing Directory Traversal Attack with a Blocked List

If you are working with file names that are ones that do not need separators, then a simple check like the one in Figure 7 can be used. This type of check in based on rejecting inputs that contain impermissible characters, so is a form of a block list. Block lists are simple to implement, so should be the simplest to get right.

The code in that figure is not difficult to fix. The attack can be prevented by replacing the check on line 4 with an explicit check for both the "/" and "\" characters. This fix works on both Unix and Windows as "/" is not a valid file name character on Windows and "\" is not a valid file name character on Unix.

---

[2] This behavior has its origins in web programming. The trailing part of a URL often is used to name a file on the web server. Since URL's use "/" as a separator, when the full file name is constructed on a Windows system, the prefix will likely have "\" as the separator and the part taken from the URL will have "/" as the separator.

12.4.2 Preventing Directory Traversal Attack with an Allowed List

Alternatively, you can describe which inputs are valid (as opposed to which are invalid). You can use regular expressions to define what characters are allowed in a file name. For example, to require that the input string consist of only alphanumeric characters, you could use this regular expression.

```
^[A-Za-z0-9]+$
```

This expression means from the start of the string "^" to the end of the string ("$"), you can have one or more ("+") of the character from the ranges A-Z, a-z, and 0-9. The Java code in Figure 8 implements this check on line 3.

However, relying on this approach only works if a simple restriction like this can be used. Unicode or other complex character encodings can easily complicate the necessary checks and create vulnerabilities. Rather than constructing paths directly with strings, when available use standard library code such as the Java Path object.

```
01 String filename = request.getParameter("file");
02 final String base = "/safedir/";
03 if(!filename.matches("[\\p{Alnum}]+")) {
04   throw new PathTraversalException(filename+" invalid");
05 }
06 File prefix = new File(base);
07 File path = new File(prefix, filename);
08 path.delete();
```

Figure 8: Code to Delete a File on the Server (Mitigation by Use of Regular Expressions)

The expression `"[\\p{Alnum}]+"` is based on the widely used regular expression library from `unicode.org`. The `\p` means a "property" in their notation and, in this case, the property is being an alphanumeric character. Alternatively, we could have used the `\w` ("word") property which include the alphanumeric characters plus the underscore character.

It is an interesting question to ask is what characters are included in the alphabetics? Before Java SDK 11, only the English letters were included. As of SDK 11 (September 2018), Java was updated to the new regular expression standard that includes all of the world's alphabets top be included in the "letter" class. This class includes both letters and ideograms (such as the Chinese characters). Most modern programming languages and libraries use this newer definition.

Regular expressions are a key technique for implemented allow and block lists and date back to the 1950's, developed by Stephen Kleene[3]. One of their earliest, and certainly most widely known uses, was in 1968 when Ken Thompson (of Unix fame) introduced them as part of the text search feature of the ed text editor (predecessor of the `ex`/`vi` and later `vim` text editors).

12.4.3 Preventing Directory Traversal Attacks with Canonical Paths

Working with canonical path names offers a more general solution since these canonical names do not include tricky references to parent directories or unresolved links. Consider the code in Figure 9 that prevents directory traversal attacks. While this code is definitely more complex and subtle than the versions in Figure 6 and Figure 7, it can prevent a variety of path traversal attacks.

The code starts by getting the file name from the user request just as in earlier examples. Next, on line 2, it creates a `String` object called `base` that contains the name of the safe directory, `/safedir`, and on line 3 it create a `File` object called `prefix` for that safe directory.

The full path name for the file to be deleted, `path`, is created on line 4, concatenating `prefix` and `filename`.

```
01 String filename = request.getParameter("file");
02 final String base = "/safedir/";
03 File prefix = new File(base);
04 File path = new File(prefix, filename);
05 // Resultant path must start with the base path prefix
06 if (!path.getCanonicalPath().startsWith(base)) {
07    throw new PathTraversalException(filename+" invalid");
08 }
09 // Resulting path must be > 1 character longer than the base
10 // path prefix
11 if (!(path.getCanonicalPath().length() >
12     prefix.getCanonicalPath().length() + 1)) {
13     throw new PathTraversalException(filename+" invalid");
14 }
15 path.delete();
```

Figure 9: Code to Delete a File on the Server (Mitigation by Use of Canonical Paths)

[3] Stephen C. Kleene, "Representation of Events in Nerve Nets and Finite Automata", in C. Shannon, E. Claude and J. McCarthy, **Automata Studies**, Princeton University Press, 1956.
Kleene was one of the founders of the University of Wisconsin-Madison Computer Sciences Department.

Before taking any further action is taken, the code makes two tests to be sure that we only delete files within the safe directory.

1. On line 6, the `if` statement checks that the safe directory `/safedir` is a prefix of the canonical path, ensuring the effective target is in that directory or underneath it in the hierarchy.
2. On line 11, the `if` statement checks that the canonical path is longer than that of the safe directory `/safedir` to ensure that we will not delete the safe directory itself. Note that the shortest valid target path name is `/safedir/X`, that is two characters longer than the prefix.

If either of these tests fails then a directory traversal attack is detected and an exception is thrown. Otherwise, the code proceeds to safely delete a file within the safe directory.

12.5 Summary

- Directory traversal attacks can occur when the attack surface reaches the construction of a path name and tricks the code into accessing an unexpected file.
- Constructed path names are manipulated in a variety of ways, including the use of "`..`", "`.`" and "/" (or in Windows "\"). Preventing these from getting into pathnames constructed from untrusted inputs minimizes the risk of a directory traversal attack.
- While there are a variety of ways to check for valid input, using canonical paths is usually the most general and reliable approach.

12.6 Exercises

1. Write a simple web service that displays the contents of text files in a directory you create. For safety, the web service should run with minimal privileges so it cannot access any important private files. The tail component of the URL path provides the file name for each request. Try to access files outside of the designated directory by using "`..`" or other tricks.
2. Modify the code in Exercise 1 to defend against directory traversal attacks, then confirm that the vulnerabilities you discovered are indeed closed. Trade places with a friend and try attacking (in the most friendly way, of course) each other's test web services.
3. Write a general subroutine that safely constructs path names from a template and one or more untrusted inputs that is conceptually similar to `printf`. Write a set of test cases that ensure it works for good inputs and also prevents directory traversal attacks.

4. Today's operating systems are still heavily influenced by the pioneering Unix file system which was designed long ago before anyone considered the possibility of directory traversal attacks. If you were designing a file system from scratch, how might you anticipate this kind of vulnerability and define path names to naturally mitigate against this kind of abuse with minimal or no mitigation required in code to prevent this kind of attack?