Introduction to Software Security

Chapter 3.3: Directory Traversal Attacks

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 1.4, March 2019.

Objectives

- Introduce basic file system concepts and terminology in order to understand the problem.
- Understand the directory traversal attack (or path traversal attack) with examples.
- Learn how to mitigate this type of an attack.

Examples in Java, however this type of attack is programming language independent.

Path names and Directory Traversal

Files are used so commonly used that often we do not give them special thought. However, careless programming can easily create a powerful vulnerability for exploit. Modern file systems are organized into directories of files and possibly more directories, forming a hierarchy where each named component has a unique path name.

A directory traversal attack (alternatively called a path traversal attack) occurs when the program constructs a path name using inputs controlled by the attacker that results in accessing an unintended file. Examples following will clarify how this works in practice.

Conceptually these attacks are similar to injection attacks in that instead of a simple identifier the attacker enters metacharacters that change the meaning of the resultant path to reference other files never intended to be possible. Any time code constructs path names to access files it is imperative to prevent malicious inputs from manipulating the resultant path in unexpected ways.

Path names consist of identifiers catenated 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. In this chapter, we will use Unix style path names but the same principles apply to both operating systems, though there are other subtle differences to consider that are beyond scope here but nonetheless important for the programmer to be aware of.

Absolute path names refer to the root of the file system and in Unix begin with a slash (‘`/`’), or in Windows begin with either ‘`\\`’ or naming a drive by letter such as ‘`C:\`’. Some absolute path name examples will clarify the various forms possible.

Unix syntax	Windows syntax
`/sample`	`\\sample`
`/dir1/file2`	`\\dir1\file2`
`/Users/bart/dev/src/ktree.c`	`C:\Users\bart\dev\src\ktree.c`

The first example above refers to the file `sample` in the root directory.

The second example refers to the file `file2` in the directory `dir1` of the root directory.

The third example refers to the file `ktree.c` in the directory `src` in the directory `dev` in the directory `bart` in the directory `Users` of the root directory (of drive C, for Windows).

Relative path names refer to the current working directory for the process context which is established by a system call (or the `cd` or `chdir` command). These paths begin with the name of a directory or file, or a single period or dot character ('.') that names the current directory at that point interpreting the path.

Unix syntax	Windows syntax
`prog.c`	`prog.c`
`./UsrGd.doc`	`.\UsrGd.doc`
`git/table.py`	`git\table.py`

The first example above refers to the file `prog.c` in the current directory.

The second example refers to the file `UsrGd.doc` in the current directory (the leading '.').

The third example refers to the file `table.py` in the directory `git` in the current directory.

One more special feature of path names completes this basic introduction. Double dot ('`..`') names the parent of the current directory, going up the file system hierarchy one level.

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

The first example above is an absolute path, interpreting it from left to right as follows: Start at the file system root that contains the `usr` directory, then the `elisa` directory therein, and then `..` refers back up

to the parent directory `usr` again; from there to the `bart` directory, the `git` directory therein, and the `gui.py` file found there.

The second example above is a relative path; the double dot ("`..`") refers to the parent of the current directory, and the directory `elisa` therein (the dot ("`.`") refers to the same directory `elisa` and could have been omitted), then finally the file therein named `table.py`.

The canonical path name is the unique path that names a file or directory that has all special names including "`.`" and "`..`", as well as symbolic links resolved. For the first example above, the canonical path name of the file is (Unix syntax) `/usr/bart/git/gui.py`

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

Web URL paths are often susceptible to directory traversal attacks as file path names because they include the same "`.`" and "`..`" syntax. This is especially true because parts of URL paths are often directly used as part of a path name.

Directory Traversal Attacks

When software builds pathname strings from inputs, attackers can influence the introduction of these various special characters (e.g., path separators, parent directory "`..`"), which provides an opportunity to introduce vulnerabilities.

To see how these attacks work, let's 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 caused the file to be deleted. Users control files in this particular directory so this should be perfectly safe.

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

The code above is invoked with a URL `file` parameter: `"../etc/passwd"`

The second line constructs a path name string `path`: `"/safedir/../etc/passwd"`

Since the directory `/safedir` is in the file system root (its parent directory) along with the `/etc` directory the effective path name is `/etc/passwd` and as a result all passwords are deleted.

The above example was much too simple, so for the next example the programmer was aware of just such an attack and they added mitigation code. (This example is directly based on an actual vulnerability discovered in our software assessment work.)

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

This code is now invoked with the same URL `file` parameter: `"../etc/passwd"`

The second line constructs a path name string `path`: `"/safedir/../etc/passwd"`

To defend against introducing parent directory references the `path` string is transformed to remove "`..`" in hopes that avoids any problem.

The third line modifies the path name string `path`: `"/safedir/etc/passwd"`

This now safely names a file under the intended directory (if it actually exists).

But there is no happy ending since attackers get second chances (and often many more). The clever attacker's next invocation provides a different URL `file` parameter: `"..../etc/passwd"`

The second line constructs a path name string `path`: `"/safedir/..../etc/passwd"`

Now the third line results in a path name string `path`: `"/safedir/../etc/passwd"`

This attempts breaks out of the intended directory and names the path `"/etc/passwd"` which results in deletion of all system 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.

Safely Mitigated Directory Traversal Attack

Working with canonical path names offers a better solution since these canonical names do not include tricky references to parent directories or unresolved links. Consider the following code that prevents directory traversal attacks.

```
String filename = request.getParameter("file");
final String base = "/safedir";
File prefix = new File(base);
File path = new File(prefix, filename);
// Resultant path must start with the base path prefix
if (!path.getCanonicalPath().startsWith(base)) {
   throw new PathTraversalException(filename + " is invalid");
}
// Resultant path must be longer by more than 1 character than the base path prefix
if (!(path.getCanonicalPath().length() > prefix.getCanonicalPath().length() + 1)) {
   throw new PathTraversalException(filename + " is invalid");
}
path.delete();
```

The code starts by getting the file name from the user request just as in earlier examples as path.

Next create a File object for the target safe directory /safedir called prefix, then create another File object relative to the safe directory for the designated file to be deleted called file.

Before taking action there are two tests to make sure we only delete files within the safe directory.

1. The first `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. The second `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 which is two characters longer than the prefix).

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

Another Subtle Kind of Attack

Here is another real-world example that further shows the risk involved in trying to clean up user input that is used to construct file names. In this case, 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. 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.

The code for this example is as follows:

```
String path = request.getParameter("file");
String fullpath = "C:\\safedir\\" + path;
//  Check for dir separators to prevent escape from safedir
If (path.contains(java.io.File.separator)) {
        throw new PathTraversalException(path + " is invalid.");
}
File f = new File(fullpath);
f.delete();
```

There is an interesting and subtle behavior of Java that subverts this approach to sanitizing the input. When Java runs on Windows, a file path name can contain either forward or backward slashes as separators. Java on Linux systems is less egalitarian; only forward slashes are valid as separators.

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 users 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
```

Note that there is an additional problem with the above code, that is when the given file name is only "..".

The good news is that the mitigation solution provided earlier will catch the problems with this current example. A call to getCanonicalPath will contain only the path separator character appropriate for the operating system on which the program is running. As a result, the comparison between the constructed string and the canonical version will fail.

Directory Traversal Mitigations

Directory traversal attacks are only possible when code builds path names based on inputs possibly manipulated by attackers, so it's safest avoiding doing such things in the first place.

If you must build path names dynamically, carefully restrict the input to characters that are safe for path components, disallowing introduced separators and path characters with special meaning. For example, requiring the input string to consist strictly of only alphanumerics, perhaps using the regular expression

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

and rejecting all others prevents directory traversal. 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.

Use library calls to transform built paths to canonical path names that can be safely checked to verify that the actual reference is to the intended portion of the file system. Converting relative paths to absolute makes it easier to examine them to ensure they are valid.

Where possible, reducing the privileges of the process that will be accessing files mitigates potential damage if the code is fooled. The attacker wants the directory traversal vulnerability to execute with administrator (super user) privileges so it can potentially access any file on the system. Reducing privilege makes this much more difficult to the extent that file access is narrowed.

It is very risky to attempt to cleverly handle all possible malicious inputs because you must correctly anticipate and block each and every possible attack of which there probably are many.

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 maliciously.
- 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.
- Checking the canonical path for the constructed path is usually the most reliable mitigation.

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?

#