Project 3: Function Interposition

Due Thursday, April 17, 2025

Introduction

We looked at the ability to do function interposition, where we can create an alternate version of a library function and force a program to load and use that version instead. This allows us to alter the behavior of library functions and, thus, the behavior of programs that use them.

This assignment requires you to write two functions that interpose two standard library functions. It comprises two parts, and each function applies to one part of the assignment.

Groups

This is an **individual** assignment. You are expected to do this independently and submit your own version of the code.

Environment

Your submission will be tested on Rutgers iLab Linux systems. You can most likely develop this on any other Linux system, but you are responsible for ensuring it will work on the iLab systems. You cannot do this assignment on macOS; the BSD functions it uses for dynamic linking are somewhat different.

Download the file `p3.zip`. It contains a `Makefile`, which compiles the code and also runs small tests, as well as an empty file `hidefile.c` for your code.

Part 1: Hiding files

A *rootkit* is malware that is designed to stay hidden on the victim's computer. If victims can't see the software, they won't know it's there.

There are various ways to hide content:

- It could be placed in obscure parts of the file system that might not be searched regularly.
- On some file systems, a file attribute can be set to be hidden, but this attribute can be changed easily.
- On Linux systems, files whose names begin with a dot are not listed by default unless you use a `-a` option to the `ls` command. This isn't a good way of hiding files since it's trivial to see them.
- The ideal way of hiding files is to modify the kernel to never report their existence, but this requires getting privileges to modify the kernel.

One way to hide files in user space is to modify the system library functions that read directory contents. This way, any programs that access directory contents will not see the file, but someone who knows it exists can open it.

In this assignment, you will use function interposition to hide the presence of files from commands such as *ls* and *find*.

Goal

The standard library function for reading directories on Linux is `readdir`, which is built on top of the `getdents` system call. Programs such as *ls*, *find*, *sh*, *zsh*, and others use `readdir` to read the contents of directories.

Your program will interpose `readdir` to hide zero or more secret files whose names are set by an environment variable. You will use Linux's `LD_PRELOAD` environment variable to load the replacement library:

```
export LD_PRELOAD=$PWD/hidefile.so
```

This tells the system to load the functions in the specified shared library before loading any others and to give the preloaded functions in this library precedence.

Your replacement function should check for file names defined in the environment variable `HIDDEN`, which will be hidden when reading a directory.

For example, you can run the command *ls* to list all files in a directory:

```
$ ls -l
total 157
-rw-r----- 1 pxk pxk 26646 Apr  2 20:39 present.pptx
-rw-r----- 1 pxk pxk    55 Apr  2 20:39 secretfile
-rw-r----- 1 pxk pxk 45224 Apr  2 20:40 secretfile.docx
-rw-r----- 1 pxk pxk  1208 Apr  2 20:40 secretfile.txt
-rw-r----- 1 pxk pxk 18792 Apr  2 20:41 status-report-1.txt
-rw-r----- 1 pxk pxk 18610 Apr  2 20:41 status-report-2.txt
-rw-r----- 1 pxk pxk 18618 Apr  2 20:41 status-report-3.txt
-rw-r----- 1 pxk pxk  3166 Apr  2 20:41 status-report-4.txt
-rw------- 1 pxk pxk   741 Apr  2 20:42 testfile.c
```

But if you set the environment variable `HIDDEN` to a specific file name, that file will no longer be visible:

```
$ export HIDDEN=secretfile.txt
$ ls -l
total 152
-rw-r----- 1 pxk pxk 26646 Apr  2 20:39 present.pptx
```

```
-rw-r----- 1 pxk pxk    55 Apr  2 20:39 secretfile
-rw-r----- 1 pxk pxk 45224 Apr  2 20:40 secretfile.docx
-rw-r----- 1 pxk pxk 18792 Apr  2 20:41 status-report-1.txt
-rw-r----- 1 pxk pxk 18610 Apr  2 20:41 status-report-2.txt
-rw-r----- 1 pxk pxk 18618 Apr  2 20:41 status-report-3.txt
-rw-r----- 1 pxk pxk  3166 Apr  2 20:41 status-report-4.txt
-rw------- 1 pxk pxk   741 Apr  2 20:42 testfile.c
```

Note that `secretfile.txt` is no longer displayed. Other commands that use *readdir*, such as *find*, will not show the file either:

```
$ find .
.
./testfile.c
./secretfile.docx
./status-report-1.txt
./secretfile
./present.pptx
./status-report-2.txt
./status-report-4.txt
./status-report-3.txt
```

Neither will using the shell's `echo` command:

```
$ echo sec*
secretfile secretfile.docx
```

Note that you'll have to start a new instance of a shell for any wildcard expansion (such as `sec*`) to hide the desired files since the original shell already loaded and linked the library functions it needs.

You can set `HIDDEN` to multiple names, separated by colon characters (`:`), to hide any files with those names

```
$ export HIDDEN=status-report-2.txt:status-report-3.txt
$ ls -l
total 115
-rw-r----- 1 pxk pxk 26646 Apr  2 20:39 present.pptx
-rw-r----- 1 pxk pxk    55 Apr  2 20:39 secretfile
-rw-r----- 1 pxk pxk 45224 Apr  2 20:40 secretfile.docx
-rw-r----- 1 pxk pxk  1208 Apr  2 20:40 secretfile.txt
-rw-r----- 1 pxk pxk 18792 Apr  2 20:41 status-report-1.txt
-rw-r----- 1 pxk pxk  3166 Apr  2 20:41 status-report-4.txt
```

```
-rw------- 1 pxk pxk   741 Apr  2 20:42 testfile.c
-rw------- 1 pxk pxk 48858 Nov 11 13:20 status-report-4.txt
-rw------- 1 pxk pxk    14 Nov 13 21:15 testfile.c
```

Note that the files `status-report-2.txt` and `status-report-3.txt` are no longer visible.

And, of course, if you delete `HIDDEN` (or set it to nothing) then all files will be visible:

```
$ unset HIDDEN
$ ls -l
total 157
-rw-r----- 1 pxk pxk 26646 Apr  2 20:39 present.pptx
-rw-r----- 1 pxk pxk    55 Apr  2 20:39 secretfile
-rw-r----- 1 pxk pxk 45224 Apr  2 20:40 secretfile.docx
-rw-r----- 1 pxk pxk  1208 Apr  2 20:40 secretfile.txt
-rw-r----- 1 pxk pxk 18792 Apr  2 20:41 status-report-1.txt
-rw-r----- 1 pxk pxk 18610 Apr  2 20:41 status-report-2.txt
-rw-r----- 1 pxk pxk 18618 Apr  2 20:41 status-report-3.txt
-rw-r----- 1 pxk pxk  3166 Apr  2 20:41 status-report-4.txt
-rw------- 1 pxk pxk   741 Apr  2 20:42 testfile.c
```

Part 2: Blocking files from opening

Another useful action is not hiding files but blocking certain files from being opened by tricking processes into thinking that they cannot access them. This is a rudimentary form of *sandboxing*, where the environment for a process is restricted beyond what the process would normally be able to access. For example, you might want to ensure that a program can open only its configuration file or that a program cannot access any files ending in `.py`.

Similar to Part 1, this part will incorporate function interposition, but this time you will prevent programs like *cat* and *file* from being able to open certain files.

Goal

The standard library function for accessing files on Linux is *open*, which acts as a wrapper for the identically named *open* system call. Programs such as *cat*, *file*, *more*, *cp*, and others call *open* on a file before they can access its contents.

Your program will interpose *open* to block any file from being opened whose name ends with any suffix contained in an environment variable named `BLOCKED`. Similar to Part 1, you will use Linux's `LD_PRELOAD` environment variable to force your *open* library function to be used instead of the standard library function:

```
export LD_PRELOAD=$PWD/hidefile.so
```

Your replacement function should check for a list of file suffixes defined in the environment variable `BLOCKED`. If the file name argument provided to the function ends with one of the listed suffixes, it will be blocked from opening.

For example, you can create a text file by running:

```
$ echo "This is sample text" > test.txt
```

Then, you can print the contents of that file by running the *cat* command:

```
$ cat test.txt
This is sample text
```

But, if you set the environment variable `BLOCKED` to the file suffix `txt`, then you will no longer be able to see the file's contents:

```
$ cat test.txt
cat: test.txt: Permission denied
```

Other commands that use open will also fail:

```
$ cp test.txt copyoftest.txt
cp: cannot open 'test.txt' for reading: Permission denied
$ file test.txt
test.txt: writable, regular file, no read permission
```

Here, you can see that the command file tells us that there is no read permission for this file, but even if you run:

```
$ chmod u+r test.txt
$ file test.txt
test.txt: writable, regular file, no read permission
```

Adding read permissions doesn't change the output because of the interposed open function. Note that the check is strictly for the suffix of a string. If you set `BLOCKED=txt`, then the files `mytxt` and `my.txt` will be blocked but `myfile.TXT` will be accessible since file names are case-sensitive. Similarly, a file named `txtfile` will be accessible since it does not end with the string "txt".

Just as in Part 1, you can set `BLOCKED` to multiple suffixes, separated by colon characters (`:`), and unset BLOCKED to prevent blocking from occurring.

What you need to do

Your assignment is to create alternate versions of the *readdir* and *open* Linux library functions.

The *readdir* function will:

1. Read the `HIDDEN` environment variable to get and parse the list of file names you need to hide.
2. Call the real version of *readdir*.
3. Check if the returned file name matches any of the names in the environment variable `HIDDEN`.
4. If it does, call *readdir* again to skip over this file entry and go to step 3.
5. Return the file data (or NULL, if there is no more data) to the calling program.

The *open* function will:

1. Read the `BLOCKED` environment variable to get and parse the list of file suffixes you need to block.
2. Check if the file path parameter ends with any of the suffixes in the environment variable `BLOCKED`.
3. If it does, then open should set `errno` to `EACCES`, the standard error code *open* returns when access of the requested file is denied and return -1.
4. If the file is not blocked, call the real version of *open*, and pass its return value to the calling program.

Multiple strings

Multiple names are delimited by colons (e.g., `file_1:file_2:file_3`) and you do not have to handle the case where the file name may contain a colon character.

LD_PRELOAD

You will use Linux's `LD_PRELOAD` mechanism. This is an environment variable that forces the listed shared libraries to be loaded for programs you run. The dynamic linker will search these libraries first when it resolves symbols in the program. This allows you to take control and replace library functions that programs use with your own versions.

Functions

For this assignment, you need to write the necessary functions in a file named `hidefile.c`. The zip file for the assignment contains a file with this name that contains a few headers and two stubbed-out functions. You might need to include more headers depending on how you parse the environment variable.

The function should be your implementations of *readdir* and *open*. You'll give them the same names, and they should take the exact same parameters as the real readdir and open functions. They will also return the same data type as the original versions. In other words, to someone calling either of the functions, it will feel just like the real function. In the case of *open*, the number of arguments is variable, so you should account for this when passing arguments to the original version.

Because you need to call the real versions of *readdir* and *open* from your functions, you will need to have your function dynamically link the original functions and pass requests to each function when needed. Read the references below for instructions on how to use the *ldsym* function to load the real version of the function from your code.

Assumptions

You may assume that the environment variables will not change once a program is running, so you can read and parse them once the first time the function is called (storing them in global or static variables so they persist).

Compilation

You will then compile this file into a shared library named `hidefile.so`. You can simply run the `make` command to do this. You can then preload this shared library by setting the environment variable:

```
    export LD_PRELOAD=$PWD/readdir.so
```

The environment variable `PWD` expands to the full pathname of the current directory, providing `LD_PRELOAD` with an absolute path. Then you can run programs such as *ls* or *cat* like in the above examples, and test whether your code works even if you change directories.

If you set `LD_PRELOAD` globally (e.g., in your login shell), don't forget to

```
unset LD_PRELOAD
```

after testing. Otherwise, this will affect any programs you run that use *readdir* or *open* differently.

Variable arguments in C

The *open* system call takes two parameters: the file name and a flag that specifies if the file will be open for reading and/or writing (and if it should be created if it doesn't exist). It also takes an optional third parameter that specifies the file's permissions. You only have to look at the pathname and can ignore the other parameters. However, you must pass them correctly to the real *open*.

Because of this optional parameter, you'll have to deal with C's ugly handling of multiple arguments – the *varargs* mechanism. The references below contain links to some tutorials, but all you need to know is that to call the real *open* function and pass it the variable number of arguments, you'll need to use code like this:

```
va_list args;
va_start(args, flags);
int fd = (*the_real_open)(pathname, flags, va_arg(args, int));
va_end(args);
```

Compiling and Testing

The assignment file p3.zip contains a placeholder for your file `hidefile.c` and a Makefile. If you run

```
make
```

the make program will compile the file `hidefile.c` into a shared library `hidefile.so`. If you run

```
make test
```

the *make* program will compile the file (if necessary), create three sample files named `secret-1.txt` , `secret-2.txt` , and `secret-3.txt`, and test your *readdir* code by setting the environment variable `HIDDEN` to a few values and then deleting it – running the *ls* command each time with the shared library preloaded.

Then, the *make* program will create three sample files named `testfile.csv`, `testfile.a`, and `testfile.b`, and test your *open* code by setting the environment variable `BLOCKED` to a few values and then deleting it – running the *cat* command each time with the shared library preloaded.

This is not an exhaustive test of your program; it's just a basic sanity test to allow you to see if it's working. You should run more tests yourself.

A note about setting LD_PRELOAD, HIDDEN, and BLOCKED

When you set environment variables in your shell, you need to export them so they will be passed to any processes created by the shell (i.e., any commands the shell runs).

For example, in *bash*, *zsh*, and *sh*, if you run

```
$ LD_PRELOAD=test.so
$ command
```

LD_PRELOAD will not preload libraries for the command since the environment setting will not be exported to child processes.

If you run:

```
$ export LD_PRELOAD=./test.so
$ command
```

Then LD_PRELOAD will be visible to all sub-processes. However, you might experience unexpected behavior since other programs (like your editor) will use the libraries that you are modifying. You will need to exit the shell or unset the variable:

```
unset LD_PRELOAD
```

Alternatively, you can set the environment variable on the command line. The shell will pass it to the command, but it will not be used in the context of your shell. This is convenient for testing:

```
LD_PRELOAD=./test.so command
```

The same applies to HIDDEN or BLOCKED. You can export them and set them to whatever you'd like:

```
$ export HIDDEN
$ HIDDEN=myfile1
$ ls -l
$ HIDDEN=myfile1:myfile2
$ ls -l
$ HIDDEN=
$ ls -l

$ export BLOCKED
$ BLOCKED=txt
$ cat test.txt
$ BLOCKED=txt:csv
$ cat test.csv
$ BLOCKED=
$ cat test.txt
```

Or set it to be effective for just one command:

```
$ HIDDEN=myfile1 ls -l
```

In this case, the shell will not set it in its environment but only for the command it runs.

Hints

The program should be quite short. My implementation was just 20-24 lines of C statements for each function.

Some of you will no doubt finish this assignment in a few minutes … but don't count on it. Others of you, however, may not have much experience with C programming or debugging in the Linux environment and have more of a struggle. Allow yourself sufficient time.

Develop in steps. Don't hesitate to put *printf* statements for debugging … but remove them before submission!

References

There are many tutorials on function interposition on the web. You'll want to follow the instructions for dynamically linking original functions as well as the proper flags to use to compile the shared library.

Some references you may find useful are:

- man page for the readdir function.
- man page for the getenv function.
- man page for ldsym.
- NetSPI Blog, Function Hooking Part I.
- CatonMat, A Simple LD_PRELOAD Tutorial, [Part 2].
- Using variable arguments in C: Tutorialspoint, gnu.org, Microsoft tutorial, jamesfisher.com

What to submit

When you have completed your assignment, you will need to submit a zip file containing ONLY `hidefile.c`. You can create this by running

```
make zip
```

which will create a zip file called `p3-submit.zip`. Validate that the file is correct before submitting.