

Project 2

Due Mar 6 by 11:59pm **Points** 10 **Submitting** a file upload **File Types** zip
Available Feb 24 at 12am - Mar 6 at 11:59pm

This assignment was locked Mar 6 at 11:59pm.

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 programs that use them.


This assignment requires you to write two small functions that interpose two standard library functions. The assignment comprises two parts. Each function is relevant to one part of the assignment.

Groups

This is an **individual** assignment. You are expected to do this on your own 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 making sure that 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 [p2.zip](https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip)  [.\(https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip\)](https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip). It contains makefiles, which compile the code and also run small tests, as well as a driver program for part 2 of this assignment. The instructions below will refer to these files.

When you unzip the file, you will see a directory called `hw-6` with three subdirectories:

`random`

contains the example of replacing the random number generator that we covered in class. The file `random.c` is the main program that prints a set of 10 random numbers and the file `myrand.c` is our

custom implementation of the *rand* library function. The `Makefile` has targets to compile both of these and run a small test. You can run

```
make
```

to compile the shared library or run

```
make test
```

to compile (if needed) and run a test. You should run this to ensure that your environment is set up correctly and you have no problems preloading libraries.



`hidefile`

contains files you need for part 1 of this assignment.

`unexpire`

contains files you need for part 2 of the assignment.

Part 1: Hiding files

In discussing malware, we covered the concept of a *rootkit*. This was malware that is designed to stay hidden on the victim's computer. If victims don't see the software then they don't know it's there.

There are various ways of hiding content. It could be placed in obscure parts of the file system that might not be searched regularly. A file might 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. Clearly, this isn't a good way of hiding files. The ideal way of hiding files is to modify the kernel to never report of their existence but this requires getting privileges to modify the kernel.

A way of hiding files in user space is to modify library functions that read directory contents. This way, programs that show directory contents will not see the file but someone who knows about it 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](https://www.man7.org/linux/man-pages/man3/readdir.3.html) (<https://www.man7.org/linux/man-pages/man3/readdir.3.html>), which is built on top of the [getdents](https://www.man7.org/linux/man-pages/man2/getdents.2.html) (<https://www.man7.org/linux/man-pages/man2/getdents.2.html>) system call. Programs such as `ls`, `find`, `sh`, `zsh`, and others use `readdir` to read contents of directories.

Your program will interpose *readdir* so that it will hide a secret file whose name is set by an environment variable. You do this by setting the `LD_PRELOAD` environment variable:

```
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 these functions precedence.

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

```
$ ls -l
total 196
-rw-r--r-- 1 pxk allusers 3855 Feb 20 18:02 present.pptx
-rw-r--r-- 1 pxk allusers 237 Feb 20 18:02 salaries.xlsx
-rw-r--r-- 1 pxk allusers 18198 Feb 20 18:02 secretfile
-rw-r--r-- 1 pxk allusers 3584 Feb 20 18:02 secretfile.docx
-rw-r--r-- 1 pxk allusers 24879 Feb 20 18:02 secretfile.txt
-rw-r--r-- 1 pxk allusers 805 Feb 20 18:01 status-report-1.txt
-rw-r--r-- 1 pxk allusers 13260 Feb 20 18:01 status-report-2.txt
-rw-r--r-- 1 pxk allusers 29878 Feb 20 18:02 status-report-3.txt
-rw-r--r-- 1 pxk allusers 5047 Feb 20 18:02 status-report-4.txt
-rw-r--r-- 1 pxk allusers 19550 Feb 20 18:03 testfile.c
```

But if you set the environment variable `HIDDEN` to a specific file name, that file will not be visible:

```
$ export HIDDEN
$ HIDDEN=secretfile.txt
$ ls -l
total 163
-rw-r--r-- 1 pxk allusers 3855 Feb 20 18:02 present.pptx
-rw-r--r-- 1 pxk allusers 237 Feb 20 18:02 salaries.xlsx
-rw-r--r-- 1 pxk allusers 18198 Feb 20 18:02 secretfile
-rw-r--r-- 1 pxk allusers 3584 Feb 20 18:02 secretfile.docx
-rw-r--r-- 1 pxk allusers 805 Feb 20 18:01 status-report-1.txt
-rw-r--r-- 1 pxk allusers 13260 Feb 20 18:01 status-report-2.txt
-rw-r--r-- 1 pxk allusers 29878 Feb 20 18:02 status-report-3.txt
-rw-r--r-- 1 pxk allusers 5047 Feb 20 18:02 status-report-4.txt
-rw-r--r-- 1 pxk allusers 19550 Feb 20 18:03 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 .
./status-report-1.txt
./present.pptx
./status-report-4.txt
./testfile.c
./status-report-3.txt
./salaries.xlsx
./secretfile
./status-report-2.txt
./secretfile.docx
```

If you set `HIDDEN` to another name then files with that name will be hidden:

```
$ HIDDEN=status-report-1.txt
$ ls -l
total 188
-rw----- 1 pxk allusers 3855 Feb 20 18:02 present.pptx
-rw----- 1 pxk allusers 237 Feb 20 18:02 salaries.xlsx
-rw----- 1 pxk allusers 18198 Feb 20 18:02 secretfile
-rw----- 1 pxk allusers 3584 Feb 20 18:02 secretfile.docx
-rw----- 1 pxk allusers 24879 Feb 20 18:02 secretfile.txt
-rw----- 1 pxk allusers 13260 Feb 20 18:01 status-report-2.txt
-rw----- 1 pxk allusers 29878 Feb 20 18:02 status-report-3.txt
-rw----- 1 pxk allusers 5047 Feb 20 18:02 status-report-4.txt
-rw----- 1 pxk allusers 19550 Feb 20 18:03 testfile.c
```

And, of course, if you delete HIDDEN then all files should be visible:



```
$ unset HIDDEN
$ ls -l
total 196
-rw----- 1 pxk allusers 3855 Feb 20 18:02 present.pptx
-rw----- 1 pxk allusers 237 Feb 20 18:02 salaries.xlsx
-rw----- 1 pxk allusers 18198 Feb 20 18:02 secretfile
-rw----- 1 pxk allusers 3584 Feb 20 18:02 secretfile.docx
-rw----- 1 pxk allusers 24879 Feb 20 18:02 secretfile.txt
-rw----- 1 pxk allusers 805 Feb 20 18:01 status-report-1.txt
-rw----- 1 pxk allusers 13260 Feb 20 18:01 status-report-2.txt
-rw----- 1 pxk allusers 29878 Feb 20 18:02 status-report-3.txt
-rw----- 1 pxk allusers 5047 Feb 20 18:02 status-report-4.txt
-rw----- 1 pxk allusers 19550 Feb 20 18:03 testfile.c
```

What you need to do

Your assignment is to create an alternate version of the *readdir* Linux library function that will:

- Call the real version of *readdir*
- Check if the returned file name matches the name in the environment variable `HIDDEN`
- If it does, call *readdir* again to skip over this file entry.
- Return the file data to the calling program.

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.

For this part, you need to write just one function in a file that you will name `hidefile.c`.

The function will be your implementation of *readdir*. You'll give it the same name and it should take the same parameters as the real [readdir](https://www.man7.org/linux/man-pages/man3/readdir.3.html) [_](https://www.man7.org/linux/man-pages/man3/readdir.3.html). It returns the same data type as the original version.

Because you need to call the real version of *readdir* from your *readdir*, you will need to have your function dynamically link the original *readdir* function and pass the request to that function. Read the

references below for instructions on how to use the *ldsym* function to load the real version of the function from your code.

You will then compile this file into a shared library named `hidefile.so`. You can then preload this shared library by setting the environment variable:

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

Then you can run programs such as *ls* or *find* as in the above examples and test whether it works.

Compiling and Testing

The assignment file [p2.zip](https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip) (<https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip>) contains a directory `hidefile`. Inside, you will find a placeholder for your `hidefile.c` code 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 files (if necessary), create two sample files named `secret-1.txt` and `secret-2.txt` and test the program by setting the environment variable `HIDDEN` to `secret-1.txt`, then `secret-2.txt`, and then deleting it – running the *ls* 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 at all.

A note about setting LD_PRELOAD and HIDDEN

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.

If you run:

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

Then `LD_PRELOAD` will be visible to all sub-processes. However, it will be also be loaded by your shell and you might experience unexpected behavior if your shell or other programs (like your editor) are using 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
```

▶ same applies to `HIDDEN`. You can export it and set it to whatever you'd like:

```
export HIDDEN  
HIDDEN=myfile1  
command  
HIDDEN=myfile2  
HIDDEN=
```

Or set it for the one command:

```
HIDDEN=myfile1 command
```

In which 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 nine lines of C statements.

Some of you will no doubt finish this assignment in well under an hour ... but don't count on it. Others of you, however, may not have much experience with C programming or 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!

Extra credit

For extra credit, you can add support to hide multiple file names.

Environment variables don't support arrays. One way to implement this would be to have a sequence of `HIDDENn` variables, such as `HIDDEN0`, `HIDDEN1`, etc. The confusion would be to know the limits and decide whether to support names such as `HIDDEN00`, `HIDDEN000`, etc.

Instead, you will implement this in a similar manner to how the shell implements specifying sequences of pathnames in search paths and library paths – by separating names with a colon. For this assignment, you can assume that file names will not contain a colon. In a real implementation, you would support an escape character for a colon.


To hide a single file, set `HIDDEN` as before:

```
HIDDEN=myfile
```

To hide multiple files, set `hidden` to a colon-separated list:

```
HIDDEN=myfile1:myfile2:myfile3:myfile4
```

Part 2

You are given a Linux program called **unexpire** that runs on the Rutgers iLab Linux systems. It's in the same [p2.zip](https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip)  [_ \(https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip\)](https://people.cs.rutgers.edu/~pxk/419/hw/files/p2.zip) .zip package as the first part of the assignment.


Imagine that this is a program that you received for evaluation and it has an expiration time coded into it. This program exits (expires) if the date is after January 1, 2022. It will also refuse to run on any date earlier than January 1, 2021. However, you want to continue using this program and you want to defeat its check for the time.

Goal

Your assignment is to replace the program's call to a system library to get the time of day with one of your own – an alternate version that will return a suitable date for the program's time check. This is a use of **function interposition**.

After the time-based check for validity is done, you want the program to use the actual time so that it will return the correct time of day.

What you need to do

This program uses the standard C library (glibc) [time](http://man7.org/linux/man-pages/man2/time.2.html)  [_ \(http://man7.org/linux/man-pages/man2/time.2.html\)](http://man7.org/linux/man-pages/man2/time.2.html) function to get the system time. The `time` function returns the number of seconds since the Linux Epoch (the start of January 1, 1970 UTC).

Your assignment is to create an alternate version of the `time()` C library function that will return a value within a time window that will pass the program's validation check.

You will use Linux's `LD_PRELOAD` mechanism to take control and replace the standard *time* library function with your own version.

You need to write just one function in a file `newtime.c`. There's a stub for this under the `unexpire` directory.

The function will be your implementation of *time*. You'll give it the same name and it should take the same parameters and return the same data type as the original version. The main program validates the time **only the first time** it calls *time*. After that, you want calls to *time* to return the correct system time. Unfortunately, your *time* function will continue to be called by the program so you will need to have your function dynamically link the original *time* function and pass successive calls to that.

You will then compile this file into a shared library called `newtime.so`. You can run

```
make
```

to compile the library. You can then preload this shared library by setting the environment variable:

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

and then run the program normally:

```
./unexpire
```

If your implementation is correct, you will see a message stating:

```
PASSED! You reset the time successfully!
```

If not, you will see a message stating what part of your implementation failed. You can also test the program by running

```
make test
```

If you set `LD_PRELOAD` globally, don't forget to

```
unset LD_PRELOAD
```

You may find that programs such as the *vi* editor call *time()* and may behave differently. See the above section, *A note about setting LD_PRELOAD and HIDDEN*, that discusses exporting an environment variable versus setting it on the same line as the command.

Hints

As with the first part, this program should be quite short: perhaps 15 lines of code if you use *strptime* to set the time from a user-friendly time string (which I recommend; it makes it easier to understand the

program and makes it more maintainable). With a hardcoded time value, your program might be down to six or lines of functional code.

Test an initial version of your program that does not link in the original *time* function just to make sure you have that first part working before you move on.

As with part 1, some of you will no doubt finish this assignment incredibly quickly while others of you might struggle a bit figuring out how to convert a date. Allow yourself sufficient time to avoid last-minute panic.

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 libraries (`hidefile.so` for part 1 and `newtime.so` for part 2).

Some reference you may find useful are:

- [Recitation notes](https://www.cs.rutgers.edu/~pxk/419/hw/files/Project-2-overview.pdf) [_ \(https://www.cs.rutgers.edu/~pxk/419/hw/files/Project-2-overview.pdf\)](https://www.cs.rutgers.edu/~pxk/419/hw/files/Project-2-overview.pdf)
- man page for the [readdir](https://www.man7.org/linux/man-pages/man3/readdir.3.html) [_ \(https://www.man7.org/linux/man-pages/man3/readdir.3.html\)](https://www.man7.org/linux/man-pages/man3/readdir.3.html) function.
- man page for the [getenv](https://www.man7.org/linux/man-pages/man3/getenv.3.html) [_ \(https://www.man7.org/linux/man-pages/man3/getenv.3.html\)](https://www.man7.org/linux/man-pages/man3/getenv.3.html) function.
- [man page](http://man7.org/linux/man-pages/man2/time.2.html) [_ \(http://man7.org/linux/man-pages/man2/time.2.html\)](http://man7.org/linux/man-pages/man2/time.2.html) for the *time* function
- man page for [ldsym](http://man7.org/linux/man-pages/man3/dlsym.3.html) [_ \(http://man7.org/linux/man-pages/man3/dlsym.3.html\)](http://man7.org/linux/man-pages/man3/dlsym.3.html).
- NetSPI Blog, [Function Hooking Part I](https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux/) [_ \(https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux/\)](https://blog.netspi.com/function-hooking-part-i-hooking-shared-library-function-calls-in-linux/)
- CatonMat, [A Simple LD_PRELOAD Tutorial](https://catonmat.net/simple-ld-preload-tutorial) [_ \(https://catonmat.net/simple-ld-preload-tutorial\)](https://catonmat.net/simple-ld-preload-tutorial)
- CatonMat, [A Simple LD_PRELOAD Tutorial, Part 2](https://catonmat.net/simple-ld-preload-tutorial-part-2) [_ \(https://catonmat.net/simple-ld-preload-tutorial-part-2\)](https://catonmat.net/simple-ld-preload-tutorial-part-2)

What to submit

When you have completed your assignment, you will need to submit a zip file containing

`hidefile/hidefile.c` and `unexpire/newtime.c`. You can create this by running

```
make zip
```

from the top-level directory (`hw-6`).

Make sure your name, netID, and RUID are in the comments of both files.

Validate that the file is correct before submitting.

