## **http://samanbarghi.com/blog/2014/09/05/how-to-wrap-a-system-call-libc-function-in-linux/**

## **Using LD\_PRELOAD**

LD\_PRELOAD allows a shared library to be loaded before any other libraries. So all I need to do is to write a shared library that overrides write and puts functions. If we wrap these functions, we need a way to call the real functions to perform the system call. dlsym just do that for us [[man 3 dlsym](http://man7.org/linux/man-pages/man3/dlsym.3.html)]: > The function dlsym() takes a “handle” of a dynamic library returned by dlopen() and the null-terminated symbol name, returning the address where that symbol is loaded into memory. If the symbol is not found, in the specified library or any of the libraries that were automatically loaded by dlopen() when that library was loaded, dlsym() returns NULL…

So inside the wrapper function we can use dlsym to get the address of the related symbol in memory and call the glibc function. Another approach can be calling the syscall directly, both approaches will work. Here is the code:

#define \_GNU\_SOURCE#include <dlfcn.h>#include <stdio.h>#include <string.h>

/\* Function pointers to hold the value of the glibc functions \*/static ssize\_t (\*real\_write)(int fd, const void \*buf, size\_t count) = NULL;static int (\*real\_puts)(const char\* str) = NULL;

/\* wrapping write function call \*/ssize\_t write(int fd, const void \*buf, size\_t count){

/\* printing out the number of characters \*/

printf("write:chars#:%lu\n", count);

/\* reslove the real write function from glibc

\* and pass the arguments.

\*/

real\_write = dlsym(RTLD\_NEXT, "write");

real\_write(fd, buf, count);

}

int puts(const char\* str){

/\* printing out the number of characters \*/

printf("puts:chars#:%lu\n", strlen(str));

/\* resolve the real puts function from glibc

\* and pass the arguments.

\*/

real\_puts = dlsym(RTLD\_NEXT, "puts");

real\_puts(str);

}

We first declare pointers to hold the value of the glibc functions, we will use these later to get the pointer from dlsym. Then we simply implement the glibc functions that we want to wrap, add our code and finally call the real function to perform the intended task.

#### **Compiling the shared library**

We compile the shared library as follows:

gcc -fPIC -shared -o bin/libpreload.so src/wrap-preload.c -ldl

We need to make sure we are generating a position-independent code(PIC) by passing -fPIC that is shared -shared. We also need to link our library with Dynamically Loaded (DL) libraries -ldl, since we are using dlsym in our code.

To run our test code and wrap glibc functions, we simply set LD\_PRELOADenviornment variable to the generated shared object file:

$ LD\_PRELOAD=/home/saman/Programming/wrap-syscall/bin/libpreload.so ./bin/testwrite:chars#:15

Hello, Kernel!puts:chars#:13

Hello, World!

LD\_PRELOAD loads the libpreload.so library before the execution of our code, and thus calling write and puts will call our wrapper functions inside the library.

## **Using ld --wrap=symbol**

Another way of wrapping functions is by using linker at the link time. GNU linker provides an option to wrap a function for a symbol [[man 1 ld](http://man7.org/linux/man-pages/man1/ld.1.html)]:

Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to “**wrap\_symbol”. Any undefined reference to “**real\_symbol” will be resolved to symbol.

This can be used to provide a wrapper for a system function. The wrapper function should be called “**wrap\_symbol”. If it wishes to call the system function, it should call “**real\_symbol”.

Here is a trivial example:

void \*

\_\_wrap\_malloc (size\_t c)

{

printf ("malloc called with %zu\n", c);

return \_\_real\_malloc (c);

}

If you link other code with this file using –wrap malloc, then all calls to “malloc” will call the function “**wrap\_malloc” instead. The call to “**real\_malloc” in “\_\_wrap\_malloc” will call the real “malloc” function.

You may wish to provide a “**real\_malloc” function as well, so that links without the –wrap option will succeed. If you do this, you should not put the definition of “**real\_malloc” in the same file as “\_\_wrap\_malloc”; if you do, the assembler may resolve the call before the linker has a chance to wrap it to “malloc”.

Based on the description, we need to implement two function \_\_real\_symbol and \_\_wrap\_symbol (in our case \_\_real\_write and \_\_wrap\_write), and link the application with our code. Here is the code:

#include <stdio.h>#include <string.h>

/\* create pointers for real glibc functions \*/

ssize\_t \_\_real\_write(int fd, const void \*buf, size\_t count);int \_\_real\_puts(const char\* str);

/\* wrapping write function \*/

ssize\_t \_\_wrap\_write (int fd, const void \*buf, size\_t count)

{

/\* printing out the number of characters \*/

printf("write:chars#:%lu\n", count);

/\* call the real glibc function and return the result \*/

ssize\_t result = \_\_real\_write(fd, buf, count);

return result;

}

/\* wrapping puts function \*/int \_\_wrap\_puts (const char\* str)

{

/\* printing out the number of characters \*/

printf("puts:chars#:%lu\n", strlen(str));

/\* call the real glibc function and return the result \*/

int result = \_\_real\_puts(str);

return result;

}

The code is very straight forward, but now lets try to compile the code and link it with our test application.

gcc -c src/wrap-link.c -o bin/wrap-link.o

gcc -c src/test.c -o bin/test-link.o

gcc -Wl,-wrap,write -Wl,-wrap=write -Wl,-wrap=puts bin/test-link.o bin/wrap-link.o -o bin/test-link-bin

I used gcc to pass the option to the linker with -Wl, which is equal to calling ldwith --wrap option. Now if I run the code I get:

$ ./bin/test-link-binwrite:chars#:15

Hello, Kernel!

puts:chars#:13

Hello, World!

## **Conclusion**

In order to wrap system calls in Linux, one have to wrap related glibc function calls. You have to be careful about the type of system calls you are tryting to override, since various functions might call different functions from glibc, e.g. printf calls putsfrom glibc which calls write at the end.

There are two ways to do this: 1-Using LD\_PRELOAD environment variable, 2-using ld --wrap. I personally prefer the first approach since if the number of wrapper functions increases I do not have to specify them one by one, as in the second case.

You can find the source code and the related Makefile in the following github repository: [https://github.com/samanbarghi/wrap-syscall](http://samanbarghi.com/blog/2014/09/05/how-to-wrap-a-system-call-libc-function-in-linux/wrap-syscall).