

How to wrap a system call (libc function) in Linux

Sep 5, 2014 · 8 minute read · 38 Comments

Linux Geeky Programming C System call



Saman Barghi

Trying to live a better life.

Blog

About

Projects



For one of my research projects I had to wrap linux system calls and redirect them to another thread. In Linux system calls are not invoked directly, but rather via wrapper functions in glibc[man 2 syscalls]. The glibc wrapper is only copying arguments and unique system call number to the registers where the kernel expects them, then trapping to kernel mode and setting the errno if the system call returns an error number [man 2 intro].

It is possible to invoke system calls directly by using syscall [man 2 syscall]. But since most programs will rely on glibc functions for system calls, it will be enough to wrap those functions. There are two ways to wrap or override C functions in Linux:

- **Using LD_PRELOAD:** There is a shell environment variable in Linux called `LD_PRELOAD`, which can be set to a path of a shared library, and that library will be loaded before any other library (including glibc).
- **Using 'ld --wrap=symbol':** This can be used to use a wrapper function for *symbol*. Any further reference to *symbol* will be resolved to the wrapper function. [man 1 ld].

I explain each approach later, but first lets write a very simple test file. I plan to wrap *write* system call and count the total number of characters that is being written out.

Test file

Lets write a very simple test file that calls *write* and *printf* to write to standard output:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    write(0, "Hello, Kernel!\n", 15);
    printf("Hello, World!\n");

    return 0;
}
```

If I run the code I get:

```
$ ./bin/test
Hello, Kernel!
Hello, World!
```

Now I want to see what are the system calls that are being called when running the test file. I use *strace* to see the system calls responsible for writting to the standard output. *strace* is being used to trace system calls and signals. Here is the result:

```
execve("./bin/test", ["./bin/test"], [/* 53 vars */]) = 0
brk(0) = 0x2532000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file
or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f099cc04000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file
or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```



Saman Barghi

Trying to live a better life.

Blog

About

Projects





Saman Barghi

Trying to live a better life.

Blog

About

Projects



```
fstat(3, {st_mode=S_IFREG|0644, st_size=128624, ...}) = 0
mmap(NULL, 128624, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f099cbe4000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file
or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\37\2\0
\0\0\0\0"... , 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1845024, ...}) = 0
mmap(NULL, 3953344, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE
, 3, 0) = 0x7f099c61e000
mprotect(0x7f099c7d9000, 2097152, PROT_NONE) = 0
mmap(0x7f099c9d9000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_
FIXED|MAP_DENYWRITE, 3, 0x1bb000) = 0x7f099c9d9000
mmap(0x7f099c9df000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_
FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f099c9df000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f099cbe3000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f099cbe1000
arch_prctl(ARCH_SET_FS, 0x7f099cbe1740) = 0
mprotect(0x7f099c9d9000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ) = 0
mprotect(0x7f099cc06000, 4096, PROT_READ) = 0
munmap(0x7f099cbe4000, 128624) = 0
write(0, "Hello, Kernel!\n", 15Hello, Kernel!) = 15
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f099cc03000
write(1, "Hello, World!\n", 14Hello, World!) = 14
exit_group(0) = ?
+++ exited with 0 +++
```

As you can see lines 26 and 29 are where the *write* system call related to our code is being called. Since our goal is to wrap glibc functions, let's check output of *ltrace* as

well. *ltrace* intercepts and records the dynamic library calls which are called by the executed process [man 1 ltrace]. Here is the result:

```
__libc_start_main(0x40057d, 1, 0x7ffffdd1ec628, 0x4005b0 <unfinishe
d ...>
write(0, "Hello, Kernel!\n", 15Hello, Kernel!
)                                     = 15
puts("Hello, World!"Hello, World!)    = 14
+++ exited (status 0) +++
```



Saman Barghi

Trying to live a better life.

Blog

About

Projects



ltrace result shows that the *write* function in the code is calling the *write* function from glibc, but *printf* is calling *puts* from glibc. So we should be careful here, overriding only the *write* function from glibc will not cause the *write* system call from *printf* to be wrapped. We need to differentiate between the final system call and the glibc library call. So in order to cover both of the cases, I need to override *write* and *puts* functions. Now let's jump into wrapping these functions.

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]: > 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:



Saman Barghi

Trying to live a better life.

Blog

About

Projects



```
#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);
    /* resolve 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_PRELOAD` environment variable to the generated shared object file:

```
$ LD_PRELOAD=/home/saman/Programming/wrap-syscall/bin/libpreload.so ./bin/test
write: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]:



Saman Barghi

Trying to live a better life.

Blog

About

Projects





Saman Barghi

Trying to live a better life.

Blog

About

Projects



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:



Saman Barghi

Trying to live a better life.

Blog

About

Projects



```
#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 `ld` with `--wrap` option. Now if I run the code I get:

```
$ ./bin/test-link-bin
write:chars#:15
Hello, Kernel!
puts:chars#:13
Hello, World!
```



Saman Barghi

Trying to live a better life.

Blog

About

Projects



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 trying to override, since various functions might call different functions from glibc, e.g. `printf` calls `puts` from 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>.

30 Comments

Saman Barghi

 Login

 Recommend 4

Sort by Best



Saman Barghi

Trying to live a better life.

Blog

About

Projects



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name



fractalspace • 3 months ago

Worked like charm! Just wondering if this `-Wl, -wrap, write` in linker is intentional? and if so, why there is no `-Wl, -wrap, puts`? (BTW the github link is dead)

^ v • Reply •



Biruk • 4 months ago

Hi My Name is Biruk and I'm Software Engineering Student. I got Question here -> how to create new directory using wrapper function (System Call) with out mkdir in C programming language is there any one to solve this problem? i can't do it alone

^ v • Reply •



Jonathan Gonzaga • 4 months ago

Hi, my name is Jonathan, I'm brazilian student. I have to port an application code from an embedded system to a Linux system. In other words, I have to take an application that runs in a microcontroller and makes that run in a Ubuntu. Such as emulator/simulator. The source code uses a RTOS (FreeRTOS) and I have to wrap some RTOS functions to Linux functions (such as `pvPortMalloc` to `malloc`). Do you believe that wrap syscalls can help me?

^ v • Reply •



Muhammad • a year ago



Saman Barghi

Trying to live a better life.

Blog

About

Projects



How to wrap a system call (libc function) in Linux · Saman Barghi

Muhammad · a year ago



Hi Saman,

I am running python script and for optimization I tried to wrap malloc function (like above) and then realized that when running the python script it did not arrive to wrap_malloc.

Do you have an Idea why, Or a way how to wrap a function that python use to allocate memory?

Thanks

^ v • Reply •



Saman Barghi Mod → Muhammad · a year ago

Hi Muhammad,

This might answer your question: [https://eklitzke.org/ld-pre...](https://eklitzke.org/ld-preload)

^ v • Reply •



Majid Rezazadeh · 2 years ago

Hi Saman

I am going to use LD_PRELOAD before Google Chrome but I receive this error:

LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.

More generally, can I use any .so file before all applications? Is there any condition to do so?

Thanks.

^ v • Reply •



Saman Barghi Mod → Majid Rezazadeh · 2 years ago

Hi Majid,

Make sure you are providing the correct path (absolute path or setting

I am not aware of any conditions for preloading libraries, and nothing is listed under `man 8 ld-linux`, so yes you should be able to preload your own libraries before the applications that your user owns (ruid == euid), e.g. you can't bypass `passwd` by a non-root user as euid is root and ruid is your userid.

^ v • Reply •



Saman Barghi

Trying to live a better life.

Blog

About

Projects



Majid Rezazadeh → Saman Barghi • 2 years ago

I think that I could find the problem! It is behind this point that it is not a dynamic executable! When I type ldd /usr/bin/chromium-browser it tells me that it is not dynamic executable. I searched for it but I couldn't find any solution till now! So, as I guessed, one of the conditions of using LD_PRELOAD before the applications is the ability of dynamic execution. Do you have any experience in this regard?

^ v • Reply •

[Show more replies](#)

[Show more replies](#)



hong • 2 years ago

Thanks for the great post! ltrace helped me out of the weird issue!

^ v • Reply •



@eckes • 2 years ago

Thanks for the article. Used it to wrap rename (btw: I added a `if (!real_...)`):

<https://gist.github.com/eck...>

^ v • Reply •

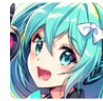


Saman Barghi Mod → @eckes • 2 years ago



Looks good :)

^ v • Reply •



re:fi.64 • 2 years ago

I have to wonder...does this work on actual *syscalls*? I mean, someone manually using the `syscall` instruction, not a C function?

^ v • Reply •

**Saman Barghi** Mod → re:fi.64 • 2 years ago

Since syscall libc functions are wrappers around system calls, to write a wrapper around the actual system call you can avoid the libc function and write your own wrapper directly. The approach above then is not applicable and does not provide any benefits. Please refer to `man 2 intro` for more information.

^ v • Reply •

**Johnson** • 3 years ago

Hi Saman,

Thanks for this great post!

I'm wondering why is it write(0, "Hello Kernel!\n", 15) when you're trying to write to the standard output? I've also tried this on my Linux and it works as well.

Can anyone explain why does "write to standard input" will be displayed on the terminal? Thanks!

^ v • Reply •

**Saman Barghi** Mod → Johnson • 3 years ago

Hi Johnson,

Thanks for your comment. I think the reason is in an interactive terminal file descriptors 0,1, and 2 are pointing to the same thing. e.g. for a pseudo terminal:

■



Saman Barghi

Trying to live a better life.

[Blog](#)[About](#)[Projects](#)

```
[~]$ ls -l /dev/fd/{0,1,2}
lrwx----- 1 saman saman 64 Jul 30 22:01 /dev/fd/0 ->
lrwx----- 1 saman saman 64 Jul 30 22:01 /dev/fd/1 ->
lrwx----- 1 saman saman 64 Jul 30 22:01 /dev/fd/2 ->
```

^ v . Reply .



novelesco → Saman Barghi • a year ago

That explains why it works. However if you intended to write to stdout then the fd should be 1.

^ v . Reply .

[Show more replies](#)



kk • 4 years ago

Hi Saman.

Thanks for this write-up. Learnt quite a bit. One question though.

We use dlsym to wrap the system-call. Is it possible to trap the system-call and do my own stuff. In the example, i would like trap the puts library call and do my own stuff instead of calling actual write system call.

^ v . Reply .



Saman Barghi Mod → kk • 4 years ago

Hi,

I am not sure if I understand the question correctly, what do you mean by trapping the system call? You can always avoid the system call and implement the wrapper function in any way you like. It is not necessary to do the actual system call, if that is your question.

^ v . Reply .



kk → Saman Barghi • 4 years ago

Hi Saman.



Saman Barghi

Trying to live a better life.

[Blog](#)

[About](#)

[Projects](#)



Yes that is my question.

^ v • Reply •



Sandhya Kumar • 4 years ago

Hi Saman,

Do you have any idea to wrap system calls like readlink(), uname()? They do not seem to have libc wrappers

^ v • Reply •



Saman Barghi Mod → Sandhya Kumar • 4 years ago

Which libc are you using? for glibc I can confirm that there exist wrappers for both of the functions you mentioned. check here: www.gnu.org/software/libc/m...

I wrote a quick test for uname, and it works without a problem.

^ v • Reply •



Sandhya Kumar → Saman Barghi • 4 years ago

There are cases which gets missed when done statically. I have raised the problem here. <http://stackoverflow.com/qu...>

^ v • Reply •



Sandhya Kumar • 4 years ago

Hi,

I am trying to hook all write() system calls being made in my "static" executable. Since linking statically, I need to go with the second approach i.e. passing wrap options to linker. This however misses some cases when checked with strace output.

For example, consider a simple program having just this line `fprintf(stderr, "Hello");`

strace output tells me that it uses `write(2, "Hello", 5);` However this does not

get hooked by my wrapper



Saman Barghi

Trying to live a better life.

Blog

About

Projects



Any pointers on how to get this working as expected?

^ v • Reply •



Saman Barghi

Trying to live a better life.

Blog

About

Projects

**Saman Barghi** Mod → Sandhya Kumar • 4 years ago

Hi Sandhya,

Usually wrapping the system call is not really wrapping the system call, but in fact wrapping the libc function that comes before it. So in your case, fprintf is causing the write system call to be called by the end, and strace shows that too. However, fprintf does not call that function directly and instead call another function from libc which make the final call to write system call. So if you do and ltrace instead of strace, you get:

```
fwrite("Hello", 1, 5, 0x7fd0feb201c0Hello)
```

so in order to wrap fprintf properly, you need to wrap fwrite instead of write. As I mentioned in the article, this is not true syscall wrap but a libc function wrap. You can read more about it here:

<http://man7.org/linux/man-p...>

^ v • Reply •

**Sandhya Kumar** → Saman Barghi • 4 years ago

Thanks! In fact, you have explained properly in "Test file" section. My bad that I missed the details.

Actually I need to hook a lot of system calls which implies I need to hook a lot lot more libc calls. Sadly, I need to do them individually as there exists no generic approach. Can you provide pointers where I can get relation (libc -> syscall) information i.e. say here write() is called by libc's write() and fwrite(). Asking this because I need an efficient way to exhaust

the list and doing strace/ltrace on every executable is naive.

^ v • Reply •



Daniel U. Thibault → Sandhya Kumar • 4 years ago

Which exact gcc or ld line did you use?

^ v • Reply •



Sandhya Kumar → Daniel U. Thibault

• 4 years ago • edited

I wrote a header file having `__real_write()` and `__wrap_write()` as mentioned in blog.

Included the header file into my code [hello.c] which just has the above mentioned `fprintf()` statement.

```
> gcc -o hello hello.c -static -Wl,--wrap=write
```

^ v • Reply •



Daniel U. Thibault • 4 years ago

The wrap-preload.c code is wrong on several levels. Have you actually tried compiling and running this? For starters, the function prototypes are:

```
static ssize_t (*real_write)(int fd, const void *buf, size_t count) = NULL;
```

and

```
static int (*real_puts)(const char *str) = NULL;
```

And `printf` calls `puts`, so preloading `libpreload.so` will result in infinite recursion.

^ v • Reply •



Saman Barghi Mod → Daniel U. Thibault • 4 years ago

Hi Daniel,

Thanks for your comment. For the first part you are absolutely right,



Saman Barghi

Trying to live a better life.

Blog

About

Projects



for some reason the astrisk character has been removed from the blog version of the code; but it is correct in the github code.
I will fix that, thanks for letting me know.

For the second part, you are absolutely right again, but there is a



Saman Barghi

Trying to live a better life.

Blog

About

Projects

