Overriding System Functions for Fun and Profit

by Ville Laurikari on Wednesday, September 23, 2009

Selectively overriding functions in shared libraries is a little known but simple enough trick. You too can replace system functions with your own versions, or hook into them to add extra functionality.

What follows works as-is on most Linux distributions. For other Unix flavors you may need to tweak a thing or four, but the general principle is the same.

Dynamic linker basics

Executable programs almost always depend on a number of shared libraries. The exception is statically linked executables, but they are nowadays exceedingly rare. You can list shared library dependencies with the `ldd` command. For example, `/bin/date` depends on a number of libraries:

```
$ ldd /bin/date
        linux-vdso.so.1 =>  (0x00007fffd51fe000)
        librt.so.1 => /lib/librt.so.1 (0x00007f0dccd6b000)
        libc.so.6 => /lib/libc.so.6 (0x00007f0dcca09000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x00007f0dcc7ed000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0dccf74000)
```

When you execute a program, the dynamic linker looks at this list of libraries. It locates the libraries on the filesystem based on configuration files and environment variables, loads the libraries into memory, links the pieces together to make a working whole, and finally executes the program.

The dynamic linker on most modern Unix flavors has a feature where you can load additional libraries to programs and selectively override functions in other shared libraries. On Linux, this feature is available via the `LD_PRELOAD` environment variable.

How to override functions

Notice "libc.so.6" on the `ldd` output? That's the C library. It provides the functions in the standard C library, such as `malloc()`, `printf()`, and `localtime()`.

To override a particular function, you simply build a shared library which exports that function. You can get a hold of the original definition of the function using `dlsym`. Here's a minimal example:

`datehack.c`:

```
#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdio.h>

struct tm *(*orig_localtime)(const time_t *timep);

struct tm *localtime(const time_t *timep)
{
  time_t t = *timep - 60 * 60 * 24;
  return orig_localtime(&t);
}

void
_init(void)
{
  printf("Loading hack.\n");
  orig_localtime = dlsym(RTLD_NEXT, "localtime");
}
```

Build this into a shared library:

```
gcc -Wall -fPIC -DPIC -c datehack.c
ld -shared -o datehack.so datehack.o -ldl
```

And we're ready to roll:

```
$ date
Wed Sep 23 18:56:08 EEST 2009
$ LD_PRELOAD=./datehack.so date
Loading hack.
Tue Sep 22 18:56:11 EEST 2009
$
```

Hey presto! When `datehack.so` is loaded, we get yesterday's time from `localtime`. As you can see, the `_init()` function is a bit special: it is called automatically when the shared library is loaded into the host process.

Some fun

In the previous post I already mentioned libtre. Here's a trick you can do to introduce approximate matching capability to any (dynamically linked) binary which uses the POSIX regex API.

First, compile libtre with the system ABI compatibility enabled:

```
wget http://laurikari.net/tre/tre-0.8.0.tar.bz2
tar xjf tre-0.8.0.tar.bz2
cd tre-0.8.0
./configure --enable-system-abi
sudo make install
```

Then load it in your favorite program. I use "less" a lot so let's use that as an example. Let's run it on the TRE README file:

```
LD_PRELOAD=/usr/local/lib/libtre.so.5 less README
```

To do a regex search, enter / followed by your regex. Try this:

```
/\<(complier){~3}\>
```

The above regex uses libtre's syntax for approximate matching to match "complier" within tree errors. The \< and \> match at the beginning and end of a word, so the regex won't match partial words.

This search turns up matches for words like "compliant", "complete", "compiled", and "compiler".

Not just a toy

There are a bunch of tools which use the LD_PRELOAD trick for something useful. Perhaps the most common use is overriding malloc(), free(), and friends to detect memory leaks and such. One of the best such tools is Valgrind. Valgrind does a whole lot more than just memory leaks, and I highly recommend it especially to C programmers.

The socksify tool intercepts calls to the connect() function (among others), and reroutes TCP connections through a SOCKS proxy. That's highly useful if your corporation is suffering from a highly paranoid IT department which allows connections only through SOCKS.

Fakeroot makes it look like you can access the filesystem as root without actually being root. This allows you to create tarballs and other packages with uid 0 files in them, without having to use root privileges.

So, there's a lot you can do with this little trick. Your imagination is the limit. What do you want to override today?

Related posts:

1. The Essence of Lambda
2. Is Your Regex Matcher Up to Snuff?

If you liked this, **click here** to receive new posts in a reader.
You should also follow me on Twitter here.

Comments on this entry are closed.

{ 9 comments }

 hackerboss September 24, 2009 at 13:20

> Overriding System Functions for Fun and Profit: http://bit.ly/mJzch

This comment was originally posted on Twitter

 lonelycoder September 25, 2009 at 13:30

I was aware of this ld_preload thing, but I though it was some kind of black magic to use it. Thanks.

 Johnny October 15, 2009 at 10:21

It's what I really need for current project. Thanks.

I need to dump all debugging messages sent to console to a file/memory. At least three solutions are available:
1. Deploy wrapped printf all over the system. Our codes are from different team/company, different wrapper functions are used to output to console, so unify them is a little difficult (even inside our team, due to historical reason, printf of system library and wrappers are mix-used, NOT well organized!)
2. Touch UART driver: it's a good solution, but we don't want it. No reason.
3. Hook into printf: I find the solution here :-)

 Johnny October 16, 2009 at 08:07

I successfully defined my own malloc/free functions, but for printf(), seems there're some problem. Have you check it? Thanks.

 Ville Laurikari October 16, 2009 at 09:39

Johnny, I think your problem might be related to a GCC optimization. GCC sometimes replaces calls to `printf()` with `puts()` as an optimization.

A simple test confirms this:

```
$ cat hello.c
#include
int main() { printf("Hello, World!\n"); return 0; }
$ gcc -o hello hello.c
$ nm --undefined-only hello
                 w _Jv_RegisterClasses
                 w __gmon_start__
                 U __libc_start_main@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
```

My code calls `printf()`, but the resulting executable actually calls `puts()`!

You need to override both `puts()` and `printf()` to catch all cases. Another option is to recompile using -fno-builtin-printf, but that's probably defeating the purpose of using LD_PRELOAD in the first place.

 AB CD October 4, 2010 at 13:00

I know this article is somewhat old, but thanks a lot, HackerBoss. And BTW, +1@lonelycoder: I must say exactly the same.

 cyro August 21, 2011 at 18:18

Is there some way to make a parasitic library which loads a target library using dlopen and maps almost all of it's symbols except for a couple of symbols and override it?

 Ville Laurikari August 21, 2011 at 21:18

@cyro, it's possible if you know the signature of each function in the target library (arguments and return type) at compile time. The implementation of your parasitic library would then define each function in the target library with an identical signature and call the target library functions (obtained with dlopen/dlsym) and return the result.

LD_PRELOAD is much simpler.

 Raman Gupta January 21, 2013 at 03:13

For anyone stumbling across this via Google looking for a mechanism to fake times (as I did), here is a complete time fake implementation using LD_PRELOAD: http://www.code-wizards.com/projects/libfaketime/index.html

{ 1 trackback }

- How to preload my .so everytime an application executes? | PHP Developer Resource May 7, 2012

Additional comments powered by BackType

Previous post: Approximate Regex Matching in Python

Next post: Putting Things in Perspective: What is Expensive, What is Cheap?

- **Get updates**
  - Subscribe to the RSS feed
  - Follow on Twitter
- **Site visitors**
- **Top Posts**

- Should You Be a Generalist Or a Specialist?
- The Birth of the Grumpy Asshole Programmer
- How to Distribute Commercial Python Applications
- Do You Make These Mistakes When Recruiting Software Developers?
- Why Your Metrics Suck
- A Developer's Most Important Interface
- Approximate Regex Matching in Python
- Why Apple Doesn't Want Your App

Recent Posts

- Do You Write Legacy Code?
- Why Apple Doesn't Want Your App
- Is Your Regex Matcher Up to Snuff?
- Metric of the Month: Duplicate Code
- Should You Be a Generalist Or a Specialist?

Twitter Updates

Error: Twitter did not respond. Please wait a few minutes and refresh this page.