“This will only hurt for a moment”: code injection on Linux and macOS with LD_PRELOAD

getambassador.io/resources/code-injection-on-linux-and-macos

Itamar Turner-Trauring

Have you ever wanted to make a program behave differently without modifying the source code? On Linux and macOS (the operating system formerly known as OS X) you can do this with the `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES` mechanisms respectively, which allow you to override the system and library calls from a particular process. While this may seem dangerous, it’s actually pretty easy to do and can be quite useful.

In this post I’ll discuss:

- Why you might want to do this.
- How these mechanisms work.
- Some limitations of the mechanisms involved, some of which impact both Linux and macOS, and some of which are limited to macOS.

Code injection for fun and profit

When you run a program it calls out to shared libraries, and to the kernel using system calls. Overriding these calls allows you to override the program’s behavior in a variety of interesting ways.

For example, your program will often need to check the current time. What if you wanted to change it to be a different value? You could change the whole system’s clock, but that’s problematic and may have unexpected side-effects on other programs.

Alternatively, you can use faketime to override the calls that retrieve the current time. Instead of getting the real time a wrapped process will get whatever time you choose to set:

```
$ /bin/date

Thu Apr 13 14:29:25 EDT 2017

$ faketime '2008-12-24 08:15:42' /bin/date

Wed Dec 24 08:15:42 EST 2008
```

Other uses include making a process look like root when setting file permissions, or pretending you have changed the root of the filesystem.

Here at Datawire, to give another example, we've been working on Telepresence, a program that makes a local process appear as if it were in a remote cluster running Kubernetes. Kubernetes runs its own DNS server, with custom domain names like `myservice.default.svc.cluster.local`, and has its own internal IPs for services. We want these IPs and DNS records to be used by the local process.

There are other ways to achieve this effect, but we've been using torsocks, which overrides TCP socket connections
and DNS lookups and routes them through a proxy. The original purpose of `torsocks` was to route calls through the Tor onion router network, which gives users greater privacy. Here you can see how my external IP changes when I run a process under `torsocks`. I send a request to ipify.org, an API that returns the callers IP address, and as you can see `torsocks` transparently routes my HTTP request through various Tor proxies:

```
$ curl http://api.ipify.org?format=json # get my external IP

{"ip":"98.216.104.162"}

$ torsocks curl http://api.ipify.org?format=json # get my external IP, via tor

{"ip":"144.217.161.119"}

$ host 144.217.161.119

119.161.217.144.in-addr.arpa domain name pointer tor-exit.clutterbuck.uk.
```

So how do all these programs work?

No process is an island

When you run a program the resulting process cannot operate on its own. It needs functionality from libraries and from the kernel; the libraries may in turn depend on other libraries or on the kernel. Consider this simple C program:

~~~~
#include <stdio.h>

int main()

{

printf("Hello, world!");

return 0;

}

```</stdio.h>

Note that this and later examples are on Linux; I'll mention differences from macOS
where relevant.

We can compile the program statically and run the resulting binary:

```console

$ gcc -static hello.c -o hello-static

$ chmod +x hello-static

$ ./hello-static

Hello, world!
~~~~

The size of the binary is rather large, considering what it does:

```
$ ls -lh hello-static

-rwxrwxr-x 1 itamarst itamarst 888K Apr 13 14:44 hello-static
```

That's because we compiled it *statically*: all the code it relies on, other than the kernel, is included in the file. We can watch calls to the kernel, aka system calls, using the `strace` utility (or `dtruss` on macOS):

```
$ strace ./hello-static > /dev/null

execve("./hello-static", ["./hello-static"], [/* 91 vars */]) = 0

... elided ...

write(1, "Hello, world!\n", 14) = 14

exit_group(0) = ?

+++ exited with 0 +++
```

As you can see the `printf` library call ended up calling the `write` system call.

From system calls to shared libraries

Most binaries are not distributed as static binaries. Instead of library code being included in the binary, the binary just notes the shared libraries it relies on, and they get loaded at runtime:

```
$ gcc -fPIC hello.c -o hello-shared

$ chmod +x hello-shared

$ ./hello-shared

Hello, world!

$ ls -lh hello-shared

-rwxrwxr-x 1 itamarst itamarst 8.4K Apr 13 14:47 hello-shared
```

We've gone from a binary of 888k to only 8k!

So what are these shared libraries the binary relies on? We can list them using `ldd` (or the similar but not identical `otool` on macOS):

```
$ ldd /bin/echo

linux-vdso.so.1 => (0x00007fff1b726000)

libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fef59330000)

/lib64/ld-linux-x86-64.so.2 (0x000056139fb3e000)
```

- `linux-vdso.so.1` is a way for the Linux kernel to inject kernel code into the process memory, so that certain system calls run faster.
- `libc.so.6` is the C standard library, which includes APIs like `printf`.
- `ld-linux-x86-64.so.2` is the dynamic linker: this is the code that knows how to load other shared libraries, like `libc.so.6`, into the process memory on startup.

Just like the static binary we can use `strace` to watch the system calls from running the shared binary:

```
$ strace ./hello-shared > /dev/null

execve("./hello-shared", ["./hello-shared"], [/* 91 vars */]) = 0

... elided ...

open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) =
832

... elided ...

write(1, "Hello, world!\n", 14) = 14

exit_group(0) = ?

+++ exited with 0 +++
```

Notice how this time the binary loaded additional files, the `libc.so.6` shared library.

In the static binary the implementation of `printf` was included in the binary itself, part of the extra 880kb of data in that binary.In the shared binary that code lives in `libc.so.6`, and we can use the `ltrace` utility to see that call:

~~~~
$ ltrace ./hello-shared > /dev/null

__libc_start_main(0x400526, 1, 0x7ffc47290b48, 0x400540 <unfinished>

puts("Hello, world!") = 14

+++ exited (status 0) +++

```</unfinished>

Where's `printf`, you may ask?

As it turns out, the generated binary isn't using `printf`, it's actually using
`puts` instead. The compiler has decided to use `puts` as an optimization since no
formatting is involved and so `puts` is a simpler and faster equivalent. We can see
that `puts` is defined but not implemented in the binary by using the `nm` utility to
look up undefined symbols:

```console

$ nm -u ./hello-shared

w __gmon_start__

w _ITM_deregisterTMCloneTable

w _ITM_registerTMCloneTable

w _Jv_RegisterClasses

U __libc_start_main@@GLIBC_2.2.5

U puts@@GLIBC_2.2.5
~~~~

Injecting shared libraries

Remember how `ld-linux` loads shared libraries? It also does some other useful things. In particular, if you set the `LD_PRELOAD` environment variable it will load the shared libraries set in that variable into the process. (On macOS this variable is called `DYLD_INSERT_LIBRARIES`.)

This injected library can override functions in other shared libraries, and if we choose call back to the original version. For example, recall that we discovered that `printf` is implemented using `puts`. Let’s override `puts` with the following shared library:

```
#include <dlfcn.h></dlfcn.h>

typedef int (*original_puts_function_type)(const char *str);

/* Our custom version that will override the libc version: */

int puts(const char *str)

{

/* Load the original puts(): */

original_puts_function_type original_puts;

original_puts = (original_puts_function_type) dlsym(RTLD_NEXT,"puts");

/* Call it twice: */

original_puts(str);

return original_puts(str);

}
```

Now we can compile this into a library, and then use it to override the call to `puts` in our shared binary:

```
$ gcc -shared -fPIC -o doubleputs.so doubleputs.c -ldl

$ LD_PRELOAD=./doubleputs.so ./hello-shared

Hello, world!

Hello, world!
```

The `ld-linux` linker loads `doubleputs.so` into the process, and all calls to `puts` get routed to our overridden version. And that's how `torsocks` and `faketime` and `fakechroot` all work: by overriding system or library calls with custom versions using the `LD_PRELOAD` mechanism.

Caveats and limitations

Code injection has its share of problems, of course.

Which functions?

Remember how we compiled a program with `printf()` but got a binary with `puts()` instead? More broadly, the library calls you need to wrap in order to inject code are hard to predict.

Some library calls will have multiple variants, some library calls will share internal private implementations with other library calls… none of them are likely to be designed for code injection.

Even worse, different operating systems and compilers will require you to wrap different calls:

- Our original example using `gcc` on Linux ended up using `puts`.
- On OS X I got a binary that called `_printf`.
- On Linux using the `clang` compiler instead of `gcc` I got a binary that called `printf`.

Static binaries and Go

Since `LD_PRELOAD` and the macOS equivalent work using the dynamic linker, it doesn't work for static binaries. Notice we don't get a double print:

```
$ LD_PRELOAD=./doubleputs.so ./hello-static

Hello, world!
```

Typically the only place you'll encounter static binaries is when writing Go. The default Go compiler has its own mechanism for calling system calls directly, and tends to ship static binaries. If you want to use `LD_PRELOAD` with Go your best bet is to use `gccgo`, the `gcc`-based Go compiler.

Security problems

For security reasons `LD_PRELOAD` doesn't work with suid binaries: the ability to inject arbitrary code into a process running as another user has some obvious problems.

On macOS there is an additional problem. Newer versions of macOS have a security subsystem called `System Integrity Protection`. For our purposes the problem is that it prevents injecting code via `DYLD_INSERT_LIBRARIES` (the macOS equivalent of `LD_PRELOAD`) into any binary in `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin`.

Luckily, there's an easy workaround. Just create a new directory, copy all the binaries from `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin` into that directory, and then add it to the start of your `$PATH` environment variable. Once the binaries are out of those special directories code injection works just fine, and since they're only 100MB copying them is quite fast.

Further reading

- Rafał Cieślak wrote an excellent intro to `LD_PRELOAD`; I borrowed the `dlsym` code from there.

- The man pages for ld-linux and dyld explain the `LD_PRELOAD` and `DYLD_INSERT_LIBRARIES` environment variables respectively in more detail.

And if you're a Kubernetes developer check out Telepresence, a great way to have a local development environment for a remote Kubernetes cluster.