My Own Private Binary

An Idiosyncratic Introduction to Linux Kernel Modules

How This Began

Several years ago, I spent a serious chunk of time figuring out how to make really teensy ELF executable files. I started down this path because I was annoyed that all of my programs, no matter how short they were, never got smaller than 4k or so. I felt that was excessive, for C, and so I started looking at what ELF files contained, and how much of that actually needed to be there. (And then, after a while, how much of it was supposed to be there but could be ripped out anyway.) Anyway, I eventually managed to shrink an executable down to 45 bytes, and I was able to demonstrate that that was the smallest possible size an ELF executable could be and still run, under x86 Linux at least.

I wrote up my findings, and some people found it interesting, and I got some positive feedback. A couple of people naturally pointed out that a shell script that did the same thing was much shorter than 45 bytes, to which my response was always that a shell script is not an executable, and if you want to consider scripts then you need to include the size of the interpreter binary along with the script size.

But then one Internet Random Person™ pointed out that I could have made a smaller executable if I had created an aout binary instead. If you don't know what aout files are, don't worry — that just means you're not old. (They are also called "a.out files", but that can be easily confused with just a file named "a.out", so I prefer to spell the name of the format "aout".) At one time the aout format was widely used on Linux, because it was Linux's only binary format. ELF was not introduced to Linux until version 2.0 (or more precisely in one of the 1.x experimental kernels). aout was, and still is, a very simple format. It sports a 32-byte header, along with a handful of other metadata. Unfortunately the aout format has some annoying limitations around dynamic linking, so the fact that Linux switched away from it early on is not too surprising. ELF is a much nicer format for a mature system. But even though aout binaries were no longer fashionable, they still worked.

However, when I first tried to run the aout executable the person had sent me, I got an "Exec format error" message — i.e. this file format is not supported. It turned out that a security issue had been uncovered at one point that involved aout core dumps. (Did you know? Executable file formats come with their own core dump file formats to match.) I'm vague on the details, but as a result most distros started compiling their kernels without aout support. The format was considered to be pretty thoroughly deprecated by that time, so it wasn't seen as a difficult call to make.

(Support for aout is still present in the kernel source tree, however, and you're free to include it if you compile your own kernel. At the time of this writing there was talk about removing it entirely, but apparently some architectures still have a use for it. Some people have suggested removing support just for aout core dumps, but in the absence of a pressing issue it remains as it is. The whole conversation is a good reminder that adding a feature to software is often far easier than removing it.)

But so I did compile a kernel with aout support, and verify that the 35-byte binary did in fact work. And the whole thing got me to wondering, *how many executable file formats does Linux actually support?* I looked into it, and I found out that the way in which Linux handles binary formats is a dynamic feature of the kernel. That is to say, it's relatively straightforward to add support for a new format, without having to recompile your kernel, or reboot your machine even.

An aside: I want to clarify that I'm not talking about the "miscellaneous binary format" feature of the kernel. That feature allows you to dynamically designate an interpreter to be run when the user attempts to execute certain files. Thus, for example, running a file ending in `.jar` can automatically invoke the JavaVM for you. That feature is controlled via the `/proc/sys/fs/binfmt_misc` system, so check out the `binfmt_misc` documentation if you're curious. Interpreters are not what I'm interested in here, though; I'm focusing exclusively on actual binary files.

What I was secretly hoping to discover was if a "flat" format existed — that is, a binary file format with no metadata at all. Obviously, such a format would allow for even smaller executables. No such format was supported, however. But that isn't too surprising, as such a format isn't very useful. Where there is no metadata, there are no features, no options. A flat format is a one-size-fits-all approach, and that's not what most people need from their binary format standards.

In order to avoid confusion, I should mention here that the Linux kernel does have support for a format that is *called* "flat", but this is just the name of the uClinux native binary format. It is actually larger and more featureful than the aout format from which it was derived. Presumably this format is flat along some other dimension.

Despite such shortcomings, it so happens that there is a flat, metadata-less executable file format that is well supported on another popular OS. If you haven't already guessed, I'm referring to the `.com` file format that MS Windows supports, having inherited it from MS-DOS, which in turn inherited it from CP/M. It is truly flat. When you run a `.com` file, the OS loads the whole thing in memory at a standard address and runs it. And this approach works okay on a single-tasking system like MS-DOS. (Or rather, on the MS-DOS-like subsystem that MS Windows presents to a running `.com` file.) In that environment, the OS can say, "Here you go, program. You have 640 kilobytes of RAM. Have fun!"

And so naturally I asked myself, *what would it take to get a `.com` file format working under Linux?* I mean yes it would not be terribly useful … but it would let me make the smallest executable file ever. Sure, I could never hope to get support for such a format added to the actual Linux kernel. But I *could* add it to my own kernel, where at least I would be able to use it myself. I could be living in my own private binary.

Kernel Modules

Linux makes it easy to do this sort of thing via *loadable kernel modules*. What exactly is a kernel module? Basically, a kernel module is an object file built for the kernel that just hasn't been linked yet. The trick is that you can link one into a running kernel without having to stop the kernel or even pull over. (This isn't quite the same thing as what is generally meant by "dynamic linking", though it is very similar in spirit.) Kernel modules mainly allow users to manage support for various kinds of hardware dynamically, but they allow you to add support for all kinds of things without having to recompile the kernel. So let's take a moment and look at how to create a kernel module.

Before we get started, we need to make sure that the kernel's header files are present on the machine. For users on Debian-based systems, this is typically done with the shell command:

`uname(1)` is used to ensure that you're getting the files that match the specific version of the kernel you're currently running.

```
$ sudo apt install linux-headers-$(uname -r)
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  linux-headers-4.15.0-156-generic
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
        ⋮
/etc/kernel/header_postinst.d/dkms:
 * dkms: running auto installation service for kernel 4.15.0-156-generic
   ...done.
```

This will install a directory under `/usr/src` corresponding to the your current kernel version. (In fact, you may find that you already have several directories located there, one for every version of the kernel that you can currently boot into.) This directory contains, among other things, all of the header files that we'll need to build kernel modules.

We'll start with a very simple one, predictably named "hello kernel".

hello.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("me");
MODULE_DESCRIPTION("hello kernel");
MODULE_VERSION("0.1");

static int __init hello_init(void)
{
    printk(KERN_INFO "hello, kernel\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "goodbye, kernel\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

A quick rundown of what is being done here:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

The header file `linux/module.h` defines the `MODULE_*` macros, `linux/init.h` defines the `__init` and `__exit` macros, and `linux/kernel.h` gives us the `printk()` function.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("me");
MODULE_DESCRIPTION("hello kernel");
MODULE_VERSION("0.1");
```

These macros just insert some metadata into our module. You can view this information via the `modprobe(1)` utility. Among other things, the kernel keeps track of the presence of non-free software.

⋮

```
module_init(hello_init);
module_exit(hello_exit);
```

And then there are two special functions, the *init* function and the *exit* function. The `module_init()` and `module_exit()` macros mark which function is which, so the kernel can find them. The first one gets called at the time the module is inserted into a running kernel, and the second one gets called when the module is being removed from the kernel.

The kernel folks have made it extremely easy to build kernel modules. Here's the makefile:

Makefile

```
obj-m = hello.o
kver = $(shell uname -r)
all:
        make -C /lib/modules/$(kver)/build/ M=$(PWD) modules
clean:
        make -C /lib/modules/$(kver)/build M=$(PWD) clean
```

All you do is put the name of your object file on the first line, and everything else is done for you.

So, if we run `make(1)`, a bunch of stuff will get created:

```
$ ls
hello.c  Makefile
$ make
make -C /lib/modules/4.15.0-156-generic/build/ M=/home/breadbox/km/hello modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-156-generic'
  CC [M]  /home/breadbox/km/hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/breadbox/km/hello/hello.mod.o
  LD [M]  /home/breadbox/km/hello/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-156-generic'
$ ls
hello.c   hello.mod.c  hello.o   modules.order
hello.ko  hello.mod.o  Makefile  Module.symvers
```

There's `hello.o`, a regular object file, but we also have an object file named `hello.ko`. This is the kernel module. We have become kernel developers.

The `insmod(8)` tool can be used to load this module into the running kernel:

```
$ sudo insmod ./hello.ko
$ lsmod | head
Module                  Size  Used by
hello                  16384  0
nls_iso8859_1          16384  0
uas                    24576  0
usb_storage            69632  1 uas
btrfs                1155072  0
zstd_compress         163840  1 btrfs
xor                    24576  1 btrfs
raid6_pq              114688  1 btrfs
ufs                    77824  0
```

When we type `lsmod(8)` to list the active modules, it's right there at the top, as the most recently added module.

The actual effect of this module is simply to output some log messages. We can use `dmesg(8)` to verify that our module really did load:

```
$ dmesg | tail -n4
[2419362.787463] e1000e: eth3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: Rx/Tx
[2448541.604799] EXT4-fs (sda1): re-mounted. Opts: (null)
[2474408.687761] usb 5-1: USB disconnect, device number 14
[2478627.755269] hello, kernel
```

We can then use `rmmod(8)` to remove the module from the running kernel whenever we want:

```
$ sudo rmmod hello
$ dmesg | tail -n4
[2448541.604799] EXT4-fs (sda1): re-mounted. Opts: (null)
[2474408.687761] usb 5-1: USB disconnect, device number 14
[2478627.755269] hello, kernel
[2478639.607702] goodbye, kernel
```

So that's all there is to writing kernel modules.

No, of course that's not true. Writing a useful kernel module does require some specialized knowledge. For example, kernel modules don't have access to `libc`, since `libc` itself is mainly an abstraction layer that sits atop the kernel. That said, much of the functionality you're used to having easy access to as a C programmer is also present in the kernel, though sometimes in slightly different dress. (Filesystems, for example, are one bit of detail that we as Unix programmers often ignore, but are obviously a major concern inside the kernel.)

But don't let all that deter you. The rewards of writing kernel modules are worth the inconveniences. There is a great deal that you can do inside a kernel module that is simply impossible outside of it. Remember, Linux is a monolithic kernel — which means that once you are loaded, you have the keys to the kingdom. Your lowly, nonstandard kernel module can do *anything*. Of course, this is a double-edged sword, because that also means that you can *accidentally* do anything. As it happens, a lot of bad things are actually pretty hard to do accidentally, such as mucking up some other process's code. You'd need to jump through a few hoops just to get a pointer to someone else's memory. But some unfortunate things are remarkably easy to do. For example, one time while working on my kernel module, I accidentally put `--i` instead of `++i` in the iterator of my `for` loop. I inserted that module into my kernel to test it, and my mouse cursor disappeared, and my music stopped playing … and then it was time to reboot my computer.

But that sort of risk shouldn't scare you away. With modern journaling file systems and the like, you're never going to be at any real risk of losing data. (I mean, unless you're actually working on implementing a filesystem, in which case please back up your files regularly.) I encourage you to experiment and try out your own ideas for kernel modules.

Binary File Formats Under Linux

All right, but what exactly does this have to do with making smaller executables? Well, as I mentioned earlier, the kernel's list of accepted binary file formats is dynamic. Specifically, this means that there are functions inside the kernel that allow code to add and remove binary formats from this list.

This is done by registering a set of callback functions, and these callbacks get invoked when the kernel is asked to execute a binary file. The kernel invokes the callbacks on this list, and the first one that claims to recognize the file takes responsibility for getting it properly loaded into memory. If nobody on the list accepts it, then the kernel sends it back to the chef with that "Exec format error" mentioned above.

Although note that at this point most shells will, as a last resort, attempt to parse the file as a shell script with no shebang line. So, depending on what the file in question actually contains, you might see a different error message entirely.

So we find ourselves in possession of the following facts:

1. The Linux kernel can dynamically introduce new binary file formats.
2. Kernel modules can be added to a running kernel.
3. Being able to run flat binaries would be really neat.

Obviously, there is only one possible response to this situation.

Version 0.1: Look Ma, No Metadata

We wish to write a kernel module that implements a flat, metadata-less binary file format for Linux. So, that's what I did.

comfile.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/mman.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/binfmts.h>
#include <linux/personality.h>
#include <linux/processor.h>
#include <linux/ptrace.h>
#include <linux/sched/task_stack.h>

MODULE_DESCRIPTION("Linux command executable files");
MODULE_AUTHOR("Brian Raiter <breadbox@muppetlabs.com>");
MODULE_VERSION("0.1");
MODULE_LICENSE("GPL");

/* Given an address or size, round up to the next page boundary.
 */
#define pagealign(n)  (((n) + PAGE_SIZE - 1) & PAGE_MASK)

static struct linux_binfmt comfile_fmt;

static int load_comfile_binary(struct linux_binprm *lbp)
{
    long const loadaddr = 0x00010000;

    char const *ext;
    loff_t filesize;
    int r;
    ext = strrchr(lbp->filename, '.');
    if (!ext || strcmp(ext, ".com"))
        return -ENOEXEC;

    r = flush_old_exec(lbp);
    if (r)
        return r;
    set_personality(PER_LINUX);
    set_binfmt(&comfile_fmt);
    setup_new_exec(lbp);

    filesize = generic_file_llseek(lbp->file, 0, SEEK_END);
    generic_file_llseek(lbp->file, 0, SEEK_SET);

    current->mm->start_code = loadaddr;
    current->mm->end_code = current->mm->start_code + filesize;

    r = setup_arg_pages(lbp, STACK_TOP, EXSTACK_DEFAULT);
    if (r)
        return r;

    r = vm_mmap(lbp->file, loadaddr, filesize,
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE, 0);
    if (r < 0)
        return r;

    install_exec_creds(lbp);
    /*finalize_exec(lbp);*/
    start_thread(current_pt_regs(), loadaddr,
                 current->mm->start_stack);
    return 0;
}

static struct linux_binfmt comfile_fmt = {
    .module = THIS_MODULE,
    .load_binary = load_comfile_binary,
    .load_shlib = NULL,
    .core_dump = NULL,
    .min_coredump = 0
};

static int __init comfile_start(void)
{
    register_binfmt(&comfile_fmt);
    return 0;
}

static void __exit comfile_end(void)
{
    unregister_binfmt(&comfile_fmt);
}

module_init(comfile_start);
module_exit(comfile_end);
```

Unlike our first kernel module, this one is actually doing some interesting work. So let's take the time to walk through this code and understand what's going on.

```
static int __init comfile_start(void)
{
    register_binfmt(&comfile_fmt);
    return 0;
}

static void __exit comfile_end(void)
{
    unregister_binfmt(&comfile_fmt);
}
```

Our very-short init function just calls `register_binfmt()`, and likewise our exit function calls `unregister_binfmt()`. As you have probably already guessed, these are the functions that add and remove support for a new binary format. The argument to both functions is a pointer to a static struct of type `linux_binfmt`.

```
static struct linux_binfmt comfile_fmt = {
    .module = THIS_MODULE,
    .load_binary = load_comfile_binary,
    .load_shlib = NULL,
    .core_dump = NULL,
```

The important fields of the `linux_binfmt` struct are three function pointers. They provide callbacks for loading an executable, loading a shared-object library, and dumping a core file. Thankfully, those latter two features are optional, so we can leave them unimplemented, and just provide the first callback.

```
        .min_coredump = 0
    };
```

```
    static int load_comfile_binary(struct linux_binprm *lbp)
```

And this function is where all the work gets done. It will be invoked by the kernel every time someone is attempting to execute one of our files, and its purpose is to get the file's contents into memory and running. The function is passed a single argument, `lbp`, which is a pointer to a struct called `linux_binprm` that contains our actual arguments. It has a dozen or so fields that summarize everything the kernel knows about our file. The callback returns an `int` value, as is typical for internal kernel functions. If all goes well, the return value is zero. When an error occurs, the function should return a negative number that corresponds to a negated `errno` value.

Recall how a program is launched under Unix: first the `fork` system call is used to duplicate the process, and then the `execve` system call replaces the process's current program with a new one.

Note that the `fork` system call is not quite the same thing as the `fork()` function supplied by `libc`, although the latter is just a thin wrapper around the former. Similarly, `libc` provides a family of seven different "exec" functions, but they all ultimately invoke the `execve` system call.

The nice thing about this system is that we never have to worry about actually creating a process from scratch. That's done for us. Every program's process is a copy of pid 1, duplicated through a succession of `fork`s. Our callback will instead be invoked during the `execve` system call. In effect, when the kernel calls us, it is asking, "Hey, I've got a file here. The user claims it's an executable binary, but it's not an ELF file. Do *you* want to deal with it?" Every callback function that has been registered with `register_binfmt()` gets called, in order, going down the list, until someone takes responsibility for the file.

So that's the first thing our callback function needs to do: it needs to decide whether or not this is actually a `.com` file. Which raises the obvious question: how do we even do that? Most binary formats looks for a magic-number identifier in the first few bytes of metadata — but we have no metadata. So then what?

Well, how does MS Windows identify `.com` files? Answer: it looks at the filename. When you try to execute a file with a name ending in "`.com`", that's all MS Windows really cares about. "Oh, you're a `.com` file, are you? Okay: here's 640k and an interrupt table. Call me when you're done."

```
        ext = strrchr(lbp->filename, '.');
        if (!ext || strcmp(ext, ".com"))
            return -ENOEXEC;
```

So that's what we do, too. One of the fields of the `linux_binprm` struct is the filename, so we examine it, and if there's no "`.com`" extension, then we return negative `ENOEXEC`, the `errno` equivalent to our "Exec format error" message. This error normally means "this is not an executable", but in this particular context, it really means "this is not one of my executables." When the kernel gets this return value, it will just continue trying other formats. If all the callbacks return this value, then `ENOEXEC` will actually get returned from `execve` itself, which `libc` will then package up and store in `errno`. But, if it does end in "`.com`", then our callback continues.

All we have to do now is load and run the file. No pressure, right? Luckily for us, the kernel provides lots of functions that will do almost all of the heavy lifting for us. We just need to oversee the whole process. So let's quickly run down the sequence of events.

```
        r = flush_old_exec(lbp);
        if (r)
            return r;
```

The very first thing we do is call `flush_old_exec()`. Boom. Nearly everything that was specific to the old process is now gone. The process is now an empty salt flat, extending featurelessly to the horizon. Wait, that's a little bleak. Instead, let's imagine it as a fallow field, ready for planting. Note also that if a non-zero value is returned, then a failure occurred, in which case we dutifully pass the negated `errno` value back up the call chain.

```
        set_personality(PER_LINUX);
```

Personality is an obscure feature that allows certain behaviors of the kernel to vary on a per-process basis. For whatever reason, it's not reset by the flush.

```
        set_binfmt(&comfile_fmt);
```

The `set_binfmt()` function explicitly claims this binary as one of our own. As far as I can tell, this is only used for debugging purposes.

```
        setup_new_exec(lbp);
```

`setup_new_exec()` initializes the process to some basic defaults, and allows for any architecture-specific initializations to occur.

```
filesize = generic_file_llseek(lbp->file, 0, SEEK_END);
generic_file_llseek(lbp->file, 0, SEEK_SET);
```

At this point we are now cleared to start defining our memory image, which is currently very empty. So the first thing we want to do is determine how big the file is, since that's also the size of program. Inside the kernel, we don't have the familiar file descriptors. Instead, we have file objects. As you might expect, the `linux_binprm` struct includes an already-opened file object, and the kernel function `generic_file_llseek()` works pretty much the same as `libc`'s more familiar `lseek()` function for retrieving the file size.

```
current->mm->start_code = loadaddr;
current->mm->end_code = current->mm->start_code + filesize;
```

`current` is a global variable that points to the current task. A task is like a process or a thread, except that instead of being a numerical identifier, it's the actual thing itself — the noumenon, the *ding-an-sich*. It's a struct with literally hundreds of fields. It's, like, really big. Pretty much anything you might want to know about a process is in this thing, somewhere. One of those things is the task's memory manager. And right now, the memory manager is eager to know where the process's component parts are going to be located. Since our format is so simple — all we have is a blob of code — we mainly need to provide a valid load address. There aren't too many requirements for this address. It just needs to be page-aligned, well away from the stack, and not zero. I selected `0x10000` as our load address because there wasn't a particular reason not to.

```
r = setup_arg_pages(lbp, STACK_TOP, EXSTACK_DEFAULT);
if (r)
    return r;
```

We aren't setting up anything else that processes typically contain, because we're just so down to earth like that, so we can go straight to calling `setup_arg_pages()`. This function finalizes the location and access permissions of the stack.

```
r = vm_mmap(lbp->file, loadaddr, filesize,
            PROT_READ | PROT_WRITE | PROT_EXEC,
            MAP_FIXED | MAP_PRIVATE, 0);
if (r < 0)
    return r;
```

And now that that's official, let's actually load something into memory. Yes folks, it's finally time to call `vm_mmap()`. This function is basically identical to `libc`'s `mmap()`, and is the natural way to load a file into (page-aligned) memory. Of course, normally when you call `mmap()` with a fixed load address you need to handle the case where that address is already in use. We don't need to worry about that here, as nothing is currently in use. We're requesting that the memory be marked as readable, writeable, and executable. Traditionally, programs will place their code into non-writeable memory, and store variable data in memory that is writeable but not executable. And that's definitely the safer way to do things, but we can't be bothered with all that. After all, the `.com` format hearkens back to a simpler time, when RAM was RAM, and didn't come with geegaws like protection. Staying true to this approach is a way of honoring our roots. Also, without metadata it's basically impossible to know which parts of the file are code and which are data, so we don't really have a choice, but it sounds better if we claim it's because of our heritage.

```
install_exec_creds(lbp);
```

We now call `install_exec_creds()`, which will set up the correct user ID vs effective user ID, in case it needs to be changed.

```
/*finalize_exec(lbp);*/
```

The function `finalize_exec()` does something with the stack's `rlimit` value. I'm a little vague on its purpose because it's somewhat new. In fact, it doesn't even exist on my kernel version, which is why it's commented out in my code. If you're running a 5.x kernel or later, feel free to restore it.

```
        start_thread(current_pt_regs(), loadaddr,
                     current->mm->start_stack);
        return 0;
```

And then, at last, we call `start_thread()`. This is the big one. We pass it a pointer to a struct that contains the process's current register values, a pointer to the top of the stack, and most importantly, the address for the instruction pointer (which for us is the same thing as the load address). The process is now ready to be scheduled. And, since we have indeed made it this far, we return zero to indicate success.

Phew. It's definitely not trivial, the process of setting up a process. But as I said, all of the real work is done by other code.

```
$ make
make -C /lib/modules/4.15.0-156-generic/build/ M=/home/breadbox/km/com modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-156-generic'
  CC [M]  /home/breadbox/km/com/comfile.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/breadbox/km/com/comfile.mod.o
  LD [M]  /home/breadbox/km/com/comfile.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-156-generic'
$ sudo insmod comfile.ko
$ lsmod | head -n3
Module                  Size  Used by
comfile                16384  0
nls_iso8859_1          16384  0
```

Now in order to actually put this kernel module to the test, we'll need a program to execute. Specifically, we need to create a binary file in our flat, metadata-less format. One that actually does something.

The down side of creating our own binary file format is that none of our usual tools know anything about it. If we want to build a program in this format, we're on our own here. But, our format is so utterly simple that this shouldn't be hard. However, it does mean that we'll need to use assembly code.

As is traditional, our minimal test program will be one that exits with a status code of 42. In order to make a system call under 64-bit Linux, we need to set `rax` to the system call number, and `rdi` to the (first) argument, and then use the `syscall` instruction. The `exit` system call is assigned the ID number 60, so this should be all we need:

```
$ cat >tiny.asm
BITS 64
        mov     rax, 60
        mov     rdi, 42
        syscall
$ nasm -f bin -o tiny.com tiny.asm
$ chmod +x tiny.com
$ wc -c tiny.com
     12 tiny.com
```

The `bin` format is `nasm`'s name for its flat binary output format, so what we get in our output file is nothing more than the assembly code that we specified.

```
$ ./tiny.com
$ echo $?
42
```

Our project has borne fruit. Behold: it works, and it's twelve bytes in size. And we can verify that it is, in fact, our kernel module that is actually loading and running it:

```
$ sudo rmmod comfile
$ ./tiny.com
bash: ./tiny.com: cannot execute binary file: Exec format error
$ sudo insmod ./comfile.ko
$ ./tiny.com
$ echo $?
42
```

This delightfully unadulterated binary file is almost a quarter the size of the smallest possible ELF executable, and less than a third the size of the aout executable that inspired this (admittedly ridiculous) exploration. And with zero bytes of overhead in our file format, we can be confident that no binary using a format that includes metadata can touch this one.

Although, of course, if we're going to start crowing about the size, then we should probably go ahead and use the smallest possible instructions.

tiny.asm

```
BITS 64
        push    42
        pop     rdi
        mov     al, 60
        syscall
```

`rdi` can be initialized in only three bytes of machine code, and `rax` can be initialized in even less, thanks to the fact that it is pre-initialized to zero.

```
$ nasm -f bin -o tiny.com tiny.asm
$ chmod +x tiny.com
$ ./tiny.com
$ echo $?
42
$ wc -c tiny.com
      7 tiny.com
```

Seven bytes. *Seven!*

To be clear, this is very much a nonstandard binary, and therefore it in no way invalidates or supplants my 45-byte ELF executable (or the aout executable). But it does make me very happy.

Further Testing of the Waters

We should try writing a few more programs, just to verify that our kernel module really does work in general. Let's try a proper hello-world program.

hello.asm

```
BITS 64

        org     0x10000

        mov     eax, 1                  ; rax = 1: write system call
        mov     edi, eax                ; rdi = 1: stdout file desc
        lea     rsi, [rel str]          ; rsi = pointer to string
        mov     edx, strlen             ; rdx = string length
        syscall                         ; call write(rdi, rsi, rdx)
        mov     eax, 60                 ; rax = 60: exit system call
        xor     edi, edi                ; rdi = 0: exit code
        syscall                         ; call exit(rdi)

str:    db      'hello, world', 10
strlen equ $ - str
```

More assembly, yes, but it's very straightforward for assembly. It compiles down to a 43-byte binary, and it does work:

```
$ nasm -f bin -o hello.com hello.asm
$ chmod +x hello.com
$ ./hello.com
hello, world
$ wc -c hello.com
     43 hello.com
```

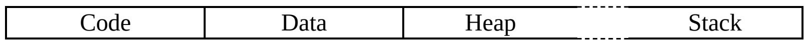
By using shorter instructions, we could reduce this program to 35 bytes, perhaps less. I will leave that as an exercise to the interested reader.

As long as our programs only use a fixed amount of data, we can allocate space by just adding it to our binary file. If we need to allocate space dynamically, however, then that's going to be a problem. Why? Because our processes don't have a heap. Why not? Because we didn't set one up in our loader. Oops.

Version 0.2: On Having a Heap

Let's address this oversight by going back to our kernel module and adding a few more lines of code. It's actually pretty easy. We just need to let the memory manager know what we want.

Most programs use a memory layout that looks something like this:



The sections are frequently broken up for the purpose of providing different access rights. Code sections are marked executable but not writeable, and the other sections are marked writeable and not executable. The code and data sections have a constant size, while the heap and the stack change in size as the program runs, with the heap growing upwards and the stack growing downwards. (And if they meet in the middle, then you've run out of memory — although on a 64-bit machine you'll run out of physical RAM long before that point.) For historical reasons, the end of the heap is called the *program break*, and the `brk` system call can be used to move it around.

If we're being pedantic, we should note that access permissions are not attached to the memory itself, but rather to the addresses. Different address ranges can be mapped to the same physical memory but with different permissions. This is not a distinction we need to worry about here.

So let's tell the memory manager that we want our processes to enjoy the benefits of a heap:

comfile.c

```
    filesize = generic_file_llseek(lbp->file, 0, SEEK_END);
    generic_file_llseek(lbp->file, 0, SEEK_SET);
    allocsize = PAGE_ALIGN(filesize);

    current->mm->start_code = loadaddr;
    current->mm->end_code = current->mm->start_code + filesize;
    current->mm->start_data = current->mm->end_code;
    current->mm->end_data = loadaddr + allocsize;
    current->mm->start_brk = current->mm->end_data;
    current->mm->brk = current->mm->start_brk;

    r = setup_arg_pages(lbp, STACK_TOP, EXSTACK_DEFAULT);
    if (r)
        return r;
```

The only measurement we have available to us is the file size, which we're using to determine the size of the code section. The `PAGE_ALIGN` macro rounds a value up to the next page boundary. Since we can't allocate a fractional number of memory pages, we can take whatever padding we'll get at the page's end, and let that be our data section. Our heap will then be located directly following this. It starts off with a size of zero, which the program can then expand as desired.

(There's no reason why we couldn't define a larger data section for our binaries, by the way. We would just need to hard-code a size for it. Perhaps, to stay true to the MS-DOS roots of our format, we should allocate a data section of 640k. But this requires making a second `vm_mmap()` call to allocate non-file-backed memory, so I've chosen to punt on that modification for the moment.)

comfile.c

```
    r = vm_mmap(lbp->file, loadaddr, filesize,
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE, 0);
    if (r < 0)
        return r;
    r = vm_brk(current->mm->start_brk, 0);
    if (r < 0)
        return r;
```

After memory has actually been allocated, we will set the process's program break to its starting value. Our process should now have a functional heap that the program can dynamically modify.

In order to test this new feature, I've written an implementation of `cat`, one that reads all of standard input into memory before doing any output. It's not an interesting program beyond being a basic demonstration of low-level heap allocation, so I won't go into it. If you're curious, you can see the source here: `cat.asm`. For now, we'll just note that it succeeds at allocating memory at runtime:

On my machine, `/etc/mailcap` is a text file well over 64k in size.

```
$ nasm -f bin -o cat.com cat.asm
$ chmod +x cat.com
$ ./cat.com </etc/mailcap | cmp - /etc/mailcap
$ wc -c cat.com
    100 cat.com
```

This version of `cat` is not really a `cat` utility, however, as it can only read from standard input. A proper `cat` program — and, indeed, a majority of proper programs — will need to be able to open files named on the command line. So how do we access the command-line arguments?

Frankly, we can't, at least not as things stand. You see, when an ELF binary runs, it has values for `argc`, `argv`, and `envp` placed at the top of its stack. Surprise! Those values are put there by the loader. Yes, this issue is also the responsiblity of our kernel module.

So let's add support for this, too.

Version 0.3: Leaving Room for Arguments

To be sure, the strings that make up the command-line arguments are present in our process's image — specifically, they're sitting in memory just above the stack. (Which means, given that the stack is currently empty, that the `rsp` register currently points to them.) But this is no neat array of string pointers. It's just a lot of strings, one after the other. A string of strings, if you will. Worse, the environment variables come immediately after the command-line arguments, without any indication of where one set ends and the other begins. So they aren't really usable, as they stand.

At the absolute least, a program needs to know how many of the strings are in each set. Well, it turns out that our kernel module has exactly that information. In the `linux_binprm` struct (provided via the argument to our callback function, remember) there are two fields named `argc` and `envc`. These are the number of command-line arguments and environment variables, respectively. In theory, if we transmit these two values to the running program, that would be enough for the code to safely access the data. Of course, if that's all we did, then every `.com` program would need to trawl through their string of strings, to determine where each item begins and ends. We could just accept that as a fact of life for programmers using our format, but since we're doing this work anyway, why not take the time to do it right? We should provide our processes with `argv` and `envp` arguments — neat arrays of pointers to the strings in question — like all the cool binary formats do.

Since these two arrays don't currently exist, we'll need to reserve some memory for them. It may feel intuitive to tap our newly-minted heap, but for this it actually makes more sense to just take it off the top of the stack. (In fact, these arrays can be seen as forming a sort of zeroth stack frame.) The top of the stack is at the top of memory, which is stored in the `linux_binprm` struct in the intuitively-named field `p`. So we want to build arrays in the memory immediately preceding this address, and then move the top of the stack down to precede our arrays.

Note, however, that we cannot use familiar functions like `strlen()` to walk through these strings. Why? Because the memory holding these strings isn't owned by the kernel; it belongs to the process itself. So far, we've only been dealing with addresses in the process's memory. We haven't tried to access that memory, except through other functions, such as `vm_mmap()`. It can thus be easy to forget that kernel memory and user memory exist in two different address spaces (not to mention different permission rings). The kernel is allowed to access user memory, of course, but it needs to be intentional about it, and this requires some extra work.

Within our kernel module, we use the `__user` annotation to declare pointers to user memory. And instead of using the `*` operator to dereference such pointers, we have two special macros: `get_user()` to read through a user pointer, and `put_user()` to write through one. And the kernel provides a handful of convenient functions, like `strnlen_user()`, that will operate on strings stored in user space.

Once we have gone through the strings and populated our two arrays, we'll still need to communicate the values for `argc`, `argv`, and `envp` to the program. The usual way to do this is to place them on the stack, allowing the program to access them at its convenience.

So let's add all this to our kernel module. We'll start by defining a separate function to handle the work of walking through the strings and building the two arrays.

comfile.c

```
/* Given argc + envc strings above the top of the stack, construct the
 * argv and envp arrays in the memory preceding, and then push argc,
 * argv, and envp onto the stack. Return the new stack top address.
 */
static unsigned long make_arrays(struct linux_binprm const *lbp)
{
    void* __user *sp;
    char* __user *argv;
    char* __user *envp;
    char __user *p;
    int i;

    p = (char __user *)lbp->p;
    envp = (char* __user *)ALIGN(lbp->p, sizeof *envp);
    envp = envp - (lbp->envc + 1);
    argv = envp - (lbp->argc + 1);
    sp = (void* __user *)argv - 3;

    current->mm->arg_start = (unsigned long)p;
    for (i = 0 ; i < lbp->argc ; ++i) {
        put_user(p, argv + i);
        p += strnlen_user(p, MAX_ARG_STRLEN);
    }
    put_user(NULL, argv + i);
    current->mm->arg_end = (unsigned long)p;

    current->mm->env_start = (unsigned long)p;
    for (i = 0 ; i < lbp->envc ; ++i) {
        put_user(p, envp + i);
        p += strnlen_user(p, MAX_ARG_STRLEN);
    }
    put_user(NULL, envp + i);
    current->mm->env_end = (unsigned long)p;

    put_user((void*)(unsigned long)lbp->argc, sp);
    put_user(argv, sp + 1);
    put_user(envp, sp + 2);
```

```
        return (unsigned long)sp;
    }
```

There's a fair bit of casting in this function because the kernel tends to store user addresses as `unsigned long` values. This may sound counterproductive, but it's somewhat natural given that kernel code rarely dereferences such addresses. But our function wants to work with them as pointer types, in order to take advantage of pointer arithmetic. (We also have to cast `argc` into a pointer while we briefly pretend that the stack is an array.)

With this function in our code, we just need to use it at the appropriate time, after the stack has been created and its address has been finalized.

comfile.c

```
        r = setup_arg_pages(lbp, STACK_TOP, EXSTACK_DEFAULT);
        if (r)
            return r;
        current->mm->start_stack = make_arrays(lbp);
```

In order to verify that all of these changes do in fact work, we can write a couple more program, `echo.asm` and `env.asm`. Again, they aren't particularly interesting in the details, so I won't dissect them here. But if you're at all familiar with reading x86 assembly, they should be relatively straightforward to understand.

```
$ nasm -f bin -o echo.com echo.asm
$ chmod +x echo.com
$ ./echo.com foo bar baz
foo bar baz
$ ./echo.com

$ wc -c echo.com
     84 echo.com
```

Take This Discussion of Practicality Outside

At this point, we have created a binary format for the Linux kernel that functions without metadata. Writing code for it is a bit of a pain, though — we have to write everything in assembly, and none of the standard tools work with our format.

Well, it so happens that there is something that we can do about that. With some investment of effort, we can coax our familiar tools into generating `.com` binaries, allowing us to use things like C compilers once more. There's a number of steps to the whole journey, however, so I've decided to put the gory details in a separate appendix, and I encourage you to peruse it if you are at all curious.

Click here to check out the appendix.

But for now, I don't want to be sidelined by meandering distractions like "usability". The focus of this essay, after all, is using kernel modules to let us produce working binaries that are *really teensy*. We have already produced a valid seven-byte executable file, and it is undeniably a thing of beauty. But a question immediately presents itself.

Could it be even smaller?

Well, we can't shrink the program itself down any further. It's as small as it can get. But maybe we could get by with a simpler program, if we changed the binary loader a little bit. Nothing too ridiculous, mind you. But I'm thinking … what if our binaries could just automatically exit when they came to the end, instead of forcing the programmer to use the `exit` system call? I mean, a majority of programming languages work that way, right? If a Python program makes it to the end of the file, it just quietly exits. Could we make our binaries do that as well?

Version 0.?: Hello, I Must Be Going

We absolutely can. What we would need to do is append a few extra bytes of machine code to the end of our file image. This code will only be executed if the program would have crashed otherwise, so one could argue that this is a purely beneficial modification to our binary format.

However, it does mean that the code size will now be larger than the file size. This alters the addresses of how we lay out memory, which could mess with programmers' assumptions, given that programs are generally written in low-level assembly.

Rather than calling this a minor feature upgrade, I therefore feel compelled to fork the code base at this point, and give this file format a new name, so as not to muddy the until-now transparent ABI of the `.com` format.

Originally, I was planning on calling this new, extended format "dot-e-com", or "dot-x-com" or something equally trite. However, after some reflection, I decided to be less boring and instead I called it the "keep-calm" (and carry on) format. This name refers naturally to the fact that you no longer have to fret about setting up mandatory instructions for exiting. Instead you can just carry on, right up to the end of the program.

However, I decided I didn't like using `.calm` as a filename extension. It was too easy to confuse with `.com` when said aloud. So I instead chose to adopt the Unicode character for a crown as the extension: . . (This is U+265A, one of the Unicode chess piece glyphs.)

Now I realize that some people may find it a bit controversial to use a non-ASCII symbol in a filename extension. But hey, the Unicode Consortium has gone to a great deal of trouble to standardize thousands upon thousands of glyphs, and we programmers continue to draw upon the same 100 or so characters for our symbols, keywords, and filenames. All these others are just sitting around, neglected. We should start using them more. I mean, why not? Heck, put emojis in your filenames. I'm not your dad.

Alan, if you're reading this: I forbid you to put emojis in your filenames.

calmfile.c

```
    ext = strrchr(lbp->filename, '.');
    ~~if (!ext || strcmp(ext, ".com"))~~
    if (!ext || strcmp(ext, ". "))
        return -ENOEXEC;
```

(If for whatever reason you don't trust your compiler to properly handle Unicode characters, you can instead assume a UTF-8 environment and write the second argument to `strcmp()` as `".\342\231\232"`.)

Anyway. The machine code that we want to append to the program can be squeezed into eight bytes:

epilog.lst

```
00000000 31FF           xor     edi, edi
00000002 678D473C       lea     eax, [edi + 60]
00000006 0F05           syscall
```

That's convenient, as it means that we can stuff it all into a `long` value, which can be stored in user memory with a simple `put_user()`. The next change we need to make is to reserve an extra eight bytes in the layout that we report to the memory manager.

calmfile.c

```
    filesize = generic_file_llseek(lbp->file, 0, SEEK_END);
    generic_file_llseek(lbp->file, 0, SEEK_SET);
    ~~allocsize = PAGE_ALIGN(filesize);~~
    codesize = filesize + 8;
    allocsize = PAGE_ALIGN(codesize);

    current->mm->start_code = loadaddr;
    ~~current->mm->end_code = current->mm->start_code + filesize;~~
    current->mm->end_code = current->mm->start_code + codesize;
    current->mm->start_data = current->mm->end_code;
    current->mm->end_data = loadaddr + allocsize;
    current->mm->start_brk = current->mm->end_data;
    current->mm->brk = current->mm->start_brk;
```

We will still use `filesize` when we call `vm_mmap()`, since we can only map what's in the file. And here we hit a subtle point. Due to the fact that memory is always mapped in page-sized chunks, we typically get more memory than we ask for from `vm_mmap()` — but not always. If the binary file happens to be exactly (or nearly exactly) page-sized, then there won't be enough memory mapped for our eight-byte epilog. So, we need to check for this edge case, and when it happens we need to call `vm_mmap()` a second time to reserve a page of anonymous memory immediately following:

calmfile.c

```
        r = vm_mmap(lbp->file, loadaddr, filesize,
                    PROT_READ | PROT_WRITE | PROT_EXEC,
                    MAP_FIXED | MAP_PRIVATE, 0);
        if (r < 0)
            return r;
        if (allocsize != PAGE_ALIGN(filesize)) {
            r = vm_mmap(NULL, loadaddr + PAGE_ALIGN(filesize), PAGE_SIZE,
                        PROT_READ | PROT_WRITE | PROT_EXEC,
                        MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, 0);
            if (r < 0)
                return r;
        }
        r = vm_brk(current->mm->start_brk, 0);
        if (r < 0)
            return r;

        put_user(0x050F3C478D67FF31, (long __user *)(loadaddr + filesize));
```

And of course that ridiculous-looking 64-bit magic number at the bottom is the little-endian encoding of our epilog.

We don't need to `rmmod` the previous module this time, since we are defining a new binary format. The two can be active simultaneously without interfering with each other.

```
$ make
make -C /lib/modules/4.15.0-156-generic/build/ M=/home/breadbox/km/calm modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-156-generic'
  CC [M]  /home/breadbox/km/calm/calmfile.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/breadbox/km/calm/calmfile.mod.o
  LD [M]  /home/breadbox/km/calm/calmfile.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-156-generic'
$ sudo insmod ./calmfile.ko
$ lsmod | head -n3
Module                  Size  Used by
calmfile               16384  0
comfile                16384  0
```

Of the programs we've currently written, `cat.asm` is the best candidate to benefit from this change, having twelve bytes' worth of machine code that can be omitted with the new format:

To enter a Unicode glyph at the terminal, type Ctrl-Shift-U, followed by the codepoint number, and then Enter. The sequence for the crown symbol is thus Ctrl-Shift-U 2 6 5 A Enter.

```
$ cp -i cat.com cat.
$ ./cat.  <hello.txt
hello, world
$ truncate -s -12 cat.
$ wc -c cat.
     88 cat.
$ ./cat.  <hello.txt
hello, world
$ echo $?
0
```

The only down side of this change is that the exit status no longer reports error values from read failures.

No, that's not true. The other, more signficant, down side of this change is that this new feature can't be used to improve the size of our seven-byte executable! That program still has to explicitly exit, in order to exit with a non-zero status. (And the non-zero exit is how we can be certain that the program actually ran, and wasn't, say, mistakenly handled as a do-nothing shell script.)

Well, it so happens that I have the perfect answer to both of these concerns.

Version 0.??: This Is Not At All a Ridiculous Idea

The way to address these issues is to modify our epilog so that instead of always returning zero, it uses the value at the top of the stack as the exit code. Additionally (and here's the brilliant part), the loader will set up our stack so that it starts out with a zero entry at the top. That way, if the program is well-behaved and pops everything off the stack that it pushed, it will automatically exit with a successful zero status — but it can also quit prematurely at any time, leaving an error code on the stack that will be automatically transmitted to the user. This is clearly an improvement to our original idea, right? I think it makes perfect sense, and isn't at all contrived.

Fortunately, this new epilog will still fit snugly into eight bytes:

epilog.lst

```
00000000 5F             pop     rdi
00000001 B83C000000     mov     eax, 60
00000006 0F05           syscall
```

This improvement requires only a three-line change to our kernel module:

```
calmfile.c

        r = vm_brk(current->mm->start_brk, 0);
        if (r < 0)
            return r;

        ~~put_user(0x050F3C478D67FF31, (long __user *)(loadaddr + filesize));~~
        put_user(0x050F0000003CB85F, (long __user *)(loadaddr + filesize));
        current->mm->start_stack -= sizeof(void*);
        put_user(0, (long __user *)current->mm->start_stack);

        install_exec_creds(lbp);
```

```
$ make
make -C /lib/modules/4.15.0-156-generic/build/ M=/home/breadbox/km/calm modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-156-generic'
  CC [M]  /home/breadbox/km/calm/calmfile.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/breadbox/km/calm/calmfile.mod.o
  LD [M]  /home/breadbox/km/calm/calmfile.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-156-generic'
$ sudo rmmod ./calmfile.ko
$ sudo insmod ./calmfile.ko
```

Once we've verified that it builds, let's revisit our current-best seven-byte binary. That same program should work just as well in our new format:

```
$ ./cat.  <tiny.asm
BITS 64
        push    42
        pop     rdi
        mov     al, 60
        syscall
$ nasm -f bin -o tiny.  tiny.asm
$ chmod +x tiny.
$ ./tiny.
$ echo $?
42
$ wc -c tiny.
      7 tiny.
```

But it should also work if we push the number 42 onto the stack and just leave it there. In other words, if the program just consists of the `push` instruction.

```
$ truncate -s 2 tiny.
$ ./tiny.
$ echo $?
42
$ wc -c tiny.
      2 tiny.
```

Two bytes. *Two bytes!* Our program is the size of a single machine-language instruction! This is the limit! There's no way to make a working program any smaller than that!

Well, I mean. I don't think there is. That is, obviously, there does exist a number that is less than two, so *in theory* it could be smaller, but how would that even be possible? Like, if there was a one-byte instruction for storing an arbitrary value on the stack, then maybe. But there isn't. The only other realistic possibility I can think of would be if the binary file could make use of some metadata to request a particular value to place on the stack initially, instead of having it be a hard-coded zero value. But there is no metadata in our file! That's the whole point of the format, right? I mean, of course, all files have some metadata; that's just the nature of filesystems. You could argue that the filename itself counts as metadata. I mean, we are basically using the filename as metadata already, I suppose. We're looking at the extension to determine the file type. So you could in theory potentially maybe argue that, just for example, using the character immediately preceding the extension would also be valid metadata, and that could be defined to specify optional behavior, like the default stack value for example …

```
calmfile.c

        put_user(0x050F0000003CB85F, (long __user *)(loadaddr + filesize));
        current->mm->start_stack -= sizeof(void*);
        ~~put_user(0, (long __user *)current->mm->start_stack);~~
        put_user(ext[-1], (long __user *)current->mm->start_stack);
```

`ext` is the pointer to the filename's extension, that we initialized way back at the top of the function, so `ext[-1]` is the character directly to the left of the dot. Don't get me wrong; I fully realize that this is indefensibly contrived — especially since, as it turns out, the ASCII character for 42 is the asterisk. That's quite an inconvenient character to have in a filename.

But having come this far, how can I *not* continue?

```
$ make
make -C /lib/modules/4.15.0-156-generic/build/ M=/home/breadbox/km/calm modules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-156-generic'
  CC [M]  /home/breadbox/km/calm/calmfile.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/breadbox/km/calm/calmfile.mod.o
  LD [M]  /home/breadbox/km/calm/calmfile.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-156-generic'
$ sudo rmmod calmfile
$ sudo insmod ./calmfile.ko
$ touch '*. '
$ chmod +x '*. '
$ './*. '
$ echo $?
42
```

Okay.

```
$ wc -c '*. '
      0 *.
```

Beat **that**, Internet Random Person™!

Illustration of author surveying the fruits of his labor by Bomberanian

(appendix)

Texts
Brian Raiter