A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux

(or, "Size *Is* Everything")

> She studied it carefully for about 15 minutes. Finally, she spoke. "There's something written on here," she said, frowning, "but it's really teensy."

[Dave Barry, "The Columnist's Caper"]

If you're a programmer who's become fed up with software bloat, then may you find herein the perfect antidote.

This document explores methods for squeezing excess bytes out of simple programs. (Of course, the more practical purpose of this document is to describe a few of the inner workings of the ELF file format and the Linux operating system. But hopefully you can also learn something about how to make really teensy ELF executables in the process.)

Please note that the information and examples given here are, for the most part, specific to ELF executables on a Linux platform running under an Intel x86 architecture. I imagine that a good bit of the information is applicable to other ELF-based Unices, but my experiences with such are too limited for me to say with certainty.

Please also note that if you aren't a little bit familiar with assembly code, you may find parts of this document sort of hard to follow. (The assembly code that appears in this document is written using Nasm; see http://www.nasm.us/.)

In order to start, we need a program. Almost any program will do, but the simpler the program the better, since we're more interested in how small we can make the executable than what the program does.

Let's take an incredibly simple program, one that does nothing but return a number back to the operating system. Why not? After all, Unix already comes with no less than two such programs: true and false. Since 0 and 1 are already taken, we'll use the number 42.

So, here is our first version:

```
/* tiny.c */
int main(void) { return 42; }
```

which we can compile and test like so:

```
$ gcc -Wall tiny.c
$ ./a.out ; echo $?
42
```

So. How big is it? Well, on my machine, I get:

```
$ wc -c a.out
   3998 a.out
```

(Yours will probably differ some.) Admittedly, that's pretty small by today's standards, but it's almost certainly bigger than it needs to be.

The obvious first step is to strip the executable:

```
$ gcc -Wall -s tiny.c
$ ./a.out ; echo $?
42
$ wc -c a.out
   2632 a.out
```

That's certainly an improvement. For the next step, how about optimizing?

```
$ gcc -Wall -s -O3 tiny.c
$ wc -c a.out
   2616 a.out
```

That also helped, but only just. Which makes sense: there's hardly anything there to optimize.

It seems unlikely that there's much else we can do to shrink a one-statement C program. We're going to have to leave C behind, and use assembler instead. Hopefully, this will cut out all the extra overhead that C programs automatically incur.

So, on to our second version. All we need to do is return 42 from main(). In assembly language, this means that the function should set the accumulator, eax, to 42, and then return:

```
; tiny.asm
BITS 32
GLOBAL main
SECTION .text
main:
              mov     eax, 42
              ret
```

We can then build and test like so:

```
$ nasm -f elf tiny.asm
$ gcc -Wall -s tiny.o
$ ./a.out ; echo $?
42
```

(Hey, who says assembly code is difficult?) And now how big is it?

```
$ wc -c a.out
   2604 a.out
```

Looks like we shaved off a measly twelve bytes. So much for all the extra overhead that C automatically incurs, eh?

Well, the problem is that we are still incurring a lot of overhead by using the main() interface. The linker is still adding an interface to the OS for us, and it is that interface that actually calls main(). So how do we get around that if we don't need it?

The actual entry point that the linker uses by default is the symbol with the name _start. When we link with gcc, it automatically includes a _start routine, one that sets up argc and argv, among other things, and then calls main().

So, let's see if we can bypass this, and define our own _start routine:

```
; tiny.asm
BITS 32
GLOBAL _start
SECTION .text
_start:
              mov     eax, 42
              ret
```

Will gcc do what we want?

```
$ nasm -f elf tiny.asm
$ gcc -Wall -s tiny.o
tiny.o(.text+0x0): multiple definition of `_start'
/usr/lib/crt1.o(.text+0x0): first defined here
/usr/lib/crt1.o(.text+0x36): undefined reference to `main'
```

No. Well, actually, yes it will, but first we need to learn how to ask for what we want.

It so happens that gcc recognizes an option called -nostartfiles. From the gcc info pages:

> -nostartfiles
> Do not use the standard system startup files when linking. The standard libraries are used normally.

Aha! Now let's see what we can do:

```
$ nasm -f elf tiny.asm
$ gcc -Wall -s -nostartfiles tiny.o
$ ./a.out ; echo $?
Segmentation fault
139
```

Well, gcc didn't complain, but the program doesn't work. What went wrong?

What went wrong is that we treated _start as if it were a C function, and tried to return from it. In reality, it's not a function at all. It's just a symbol in the object file which the linker uses to locate the program's entry point. When our program is invoked, it's invoked directly. If we were to look, we would see that the value on the top of the stack was the number 1, which is certainly very un-address-like. In fact, what is on the stack is our program's argc value. After this comes the elements of the argv array, including the terminating NULL element, followed by the elements of envp. And that's all. There is no return address on the stack.

So, how does _start ever exit? Well, it calls the exit() function! That's what it's there for, after all.

Actually, I lied. What it really does is call the _exit() function. (Notice the leading underscore.) exit() is required to finish up some tasks on behalf of the process, but those tasks will never have been started, because we're bypassing the library's startup code. So we need to bypass the library's shutdown code as well, and go directly to the operating system's shutdown processing.

So, let's try this again. We're going to call _exit(), which is a function that takes a single integer argument. So all we need to do is push the number onto the stack and call the function. (We also need to declare _exit() as external.) Here's our assembly:

```
; tiny.asm
BITS 32
```

```
EXTERN _exit
GLOBAL _start
SECTION .text
_start:
              push    dword 42
              call    _exit
```

And we build and test as before:

```
$ nasm -f elf tiny.asm
$ gcc -Wall -s -nostartfiles tiny.o
$ ./a.out ; echo $?
42
```

Success at last! And now how big is it?

```
$ wc -c a.out
   1340 a.out
```

Almost half the size! Not bad. Not bad at all. Hmmm ... so what other interesting obscure options does gcc have?

Well, this one, appearing immediately after -nostartfiles in the documentation, is certainly eye-catching:

> -nostdlib
> Don't use the standard system libraries and startup files when linking. Only the files you specify will be passed to the linker.

That's gotta be worth investigating:

```
$ gcc -Wall -s -nostdlib tiny.o
tiny.o(.text+0x6): undefined reference to `_exit'
```

Oops. That's right ... _exit() is, after all, a library function. It has to be filled in from somewhere.

Okay. But surely, we don't need libc's help just to end a program, do we?

No, we don't. If we're willing to leave behind all pretenses of portability, we can make our program exit without having to link with anything else. First, though, we need to know how to make a system call under Linux.

Linux, like most operating systems, provides basic necessities to the programs it hosts via system calls. This includes things like opening a file, reading and writing to file handles — and, of course, shutting down a process.

The Linux system call interface is a single instruction: int 0x80. All system calls are done via this interrupt. To make a system call, eax should contain a number that indicates which system call is being invoked, and other registers are used to hold the arguments, if any. If the system call takes one argument, it will be in ebx; a system call with two arguments will use ebx and ecx. Likewise, edx, esi, and edi are used if a third, fourth, or fifth argument is required, respectively. Upon return from a system call, eax will contain the return value. If an error occurs, eax will contain a negative value, with the absolute value indicating the error.

The numbers for the different system calls are listed in /usr/include/asm/unistd.h. A quick peek will tell us that the exit system call is assigned the number 1. Like the C function, it takes one argument, the value to return to the parent process, and so this will go into ebx.

We now know all we need to know to create the next version of our program, one that won't need assistance from any external functions to work:

```
; tiny.asm
BITS 32
GLOBAL _start
SECTION .text
_start:
              mov     eax, 1
              mov     ebx, 42
              int     0x80
```

Here we go:

```
$ nasm -f elf tiny.asm
$ gcc -Wall -s -nostdlib tiny.o
$ ./a.out ; echo $?
42
```

Ta-da! And the size?

```
$ wc -c a.out
    372 a.out
```

Now *that's* tiny! Almost a fourth the size of the previous version!

So ... can we do anything else to make it even smaller?

How about using shorter instructions?

If we generate a list file for the assembly code, we'll find the following:

```
00000000 B801000000        mov        eax, 1
00000005 BB2A000000        mov        ebx, 42
0000000A CD80              int        0x80
```

Well, gee, we don't need to initialize all of ebx, since the operating system is only going to use the lowest byte. Setting bl alone will be sufficient, and will take two bytes instead of five.

We can also set eax to one by xor'ing it to zero and then using a one-byte increment instruction; this will save two more bytes.

```
00000000 31C0              xor        eax, eax
00000002 40                inc        eax
00000003 B32A              mov        bl, 42
00000005 CD80              int        0x80
```

I think it's pretty safe to say that we're not going to make this program any smaller than that.

As an aside, we might as well stop using gcc to link our executable, seeing as we're not using any of its added functionality, and just call the linker, ld, ourselves:

```
$ nasm -f elf tiny.asm
$ ld -s tiny.o
$ ./a.out ; echo $?
42
```

```
$ wc -c a.out
    368 a.out
```

Four bytes smaller. (Hey! Didn't we shave five bytes off? Well, we did, but alignment considerations within the ELF file caused it to require an extra byte of padding.)

So ... have we reached the end? Is this as small as we can go?

Well, hm. Our program is now seven bytes long. Do ELF files really require 361 bytes of overhead? What's in this file, anyway?

We can peek into the contents of the file using objdump:

```
$ objdump -x a.out | less
```

The output may look like gibberish, but right now let's just focus on the list of sections:

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000007  08048080  08048080  00000080  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .comment      0000001c  00000000  00000000  00000087  2**0
                  CONTENTS, READONLY
```

The complete .text section is listed as being seven bytes long, just as we specified. So it seems safe to conclude that we now have complete control of the machine-language content of our program.

But then there's this other section named ".comment". Who ordered *that*? And it's 28 bytes long, even! We may not be sure what this .comment section is, but it seems a good bet that it isn't a necessary feature....

The .comment section is listed as being located at file offset 00000087 (hexadecimal). If we use a hexdump program to look at that area of the file, we will see:

```
00000080: 31C0 40B3 2ACD 8000 5468 6520 4E65 7477  1.@.*...The Netw
00000090: 6964 6520 4173 7365 6D62 6C65 7220 302E  ide Assembler 0.
000000A0: 3938 0000 2E73 796D 7461 6200 2E73 7472  98...symtab..str
```

Well, well, well. Who'd've thought that Nasm would undermine our quest like this? Maybe we should switch to using gas, AT&T syntax notwithstanding....

Alas, if we do:

```
; tiny.s
.globl _start
.text
_start:
              xorl    %eax, %eax
              incl    %eax
              movb    $42, %bl
              int     $0x80
```

... we will find:

```
$ gcc -s -nostdlib tiny.s
$ ./a.out ; echo $?
42
```

```
$ wc -c a.out
    368 a.out
```

... no difference!

Well, actually there is some difference. Turning once again to objdump, we see:

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000007  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0804907c  0804907c  0000007c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0804907c  0804907c  0000007c  2**2
                  ALLOC
```

No comment section, but now we have two useless sections for storing our nonexistent data. And even though these sections are zero bytes long, they incur overhead, bringing our file size up for no good reason.

Okay, so just what is all this overhead, and how do we get rid of it?

Well, to answer these questions, we must begin diving into some real wizardry. We need to understand the ELF format.

The canonical document describing the ELF format for Intel-386 architectures can be found at http://refspecs.linuxbase.org/elf/elf.pdf. (You can also find a flat-text version of version 1.0 of the standard at http://www.muppetlabs.com/~breadbox/software/ELF.txt.) This specification covers a lot of territory, so if you'd prefer to not read the whole thing yourself, I'll understand. Basically, here's what we need to know:

Every ELF file begins with a structure called the ELF header. This structure is 52 bytes long, and contains several pieces of information that describe the contents of the file. For example, the first sixteen bytes contain an "identifier", which includes the file's magic-number signature (7F 45 4C 46), and some one-byte flags indicating that the contents are 32-bit or 64-bit, little-endian or big-endian, etc. Other fields in the ELF header contain information such as: the target architecture; whether the ELF file is an executable, an object file, or a shared-object library; the program's starting address; and the locations within the file of the program header table and the section header table.

These two tables can appear anywhere in the file, but typically the former appears immediately following the ELF header, and the latter appears at or near the end of the file. The two tables serve similar purposes, in that they identify the component parts of the file. However, the section header table focuses more on identifying where the various parts of the program are within the file, while the program header table describes where and how these parts are to be loaded into memory. In brief, the section header table is for use by the compiler and linker, while the program header table is for use by the program loader. The program header table is optional for object files, and in practice is never present. Likewise, the section header table is optional for executables — but is almost *always* present!

So, this is the answer to our first question. A fair piece of the overhead in our program is a completely unnecessary section header table, and maybe some equally useless sections that don't contribute to our program's memory image.

So, we turn to our second question: how do we go about getting rid of all that?

Alas, we're on our own here. None of the standard tools will deign to make an executable without a section header table of some kind. If we want such a thing, we'll have to do it ourselves.

This doesn't quite mean that we have to pull out a binary editor and code the hexadecimal values by hand, though. Good old Nasm has a flat binary output format, which will serve us well. All we need now is the image of an empty ELF executable, which we can fill in with our program. Our program, and nothing else.

We can look at the ELF specification, and /usr/include/linux/elf.h, and executables created by the standard tools, to figure out what our empty ELF executable should look like. But, if you're the impatient type, you can just use the one I've supplied here:

```
BITS 32

              org     0x08048000

ehdr:                                                 ; Elf32_Ehdr
              db      0x7F, "ELF", 1, 1, 1, 0         ;   e_ident
      times 8 db      0
              dw      2                               ;   e_type
              dw      3                               ;   e_machine
              dd      1                               ;   e_version
              dd      _start                          ;   e_entry
              dd      phdr - $$                       ;   e_phoff
              dd      0                               ;   e_shoff
              dd      0                               ;   e_flags
              dw      ehdrsize                        ;   e_ehsize
              dw      phdrsize                        ;   e_phentsize
              dw      1                               ;   e_phnum
              dw      0                               ;   e_shentsize
              dw      0                               ;   e_shnum
              dw      0                               ;   e_shstrndx

ehdrsize      equ     $ - ehdr

phdr:                                                 ; Elf32_Phdr
              dd      1                               ;   p_type
              dd      0                               ;   p_offset
              dd      $$                              ;   p_vaddr
              dd      $$                              ;   p_paddr
              dd      filesize                        ;   p_filesz
              dd      filesize                        ;   p_memsz
              dd      5                               ;   p_flags
              dd      0x1000                          ;   p_align

phdrsize      equ     $ - phdr

_start:

; your program here

filesize      equ     $ - $$
```

This image contains an ELF header, identifying the file as an Intel 386 executable, with no section header table and a program header table containing one entry. Said entry instructs the program loader to load the entire file into memory (it's normal behavior for a program to include its ELF header and program header table in its memory image) starting at memory address 0x08048000 (which is the default address for executables to load), and to begin executing the code at _start, which appears immediately after the program header table. No .data segment, no .bss segment, no commentary — nothing but the bare necessities.

So, let's add in our little program:

```
; tiny.asm
              org     0x08048000

;
; (as above)
;



_start:
              mov     bl, 42
              xor     eax, eax
              inc     eax
              int     0x80

filesize      equ     $ - $$
```

and try it out:

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
```

We have just created an executable completely from scratch. How about that? And now, take a look at its size:

```
$ wc -c a.out
     91 a.out
```

Ninety-one bytes. Less than one-fourth the size of our previous attempt, and less than one-*fortieth* the size of our first!

What's more, this time we can account for every last byte. We know exactly what's in the executable, and why it needs to be there. This is, finally, the limit. We can't get any smaller than this.

Or can we?

Well, if you actually stopped to read the ELF specification, you might have noticed a couple of facts. 1) The different parts of an ELF file are permitted to be located anywhere (except the ELF header, which must be at the top of the file), and they can even overlap each other. 2) Some of the fields in the headers aren't actually used.

In particular, I'm thinking of that string of zeros at the end of the 16-byte identification field. They are pure padding, to make room for future expansion of the ELF standard. So the OS shouldn't care at all what's in there. And we're already loading everything into memory anyway, and our program is only seven bytes long....

Can we put our code inside the ELF header itself?

Why not?

```
; tiny.asm

BITS 32

              org     0x08048000

ehdr:                                                 ; Elf32_Ehdr
              db      0x7F, "ELF"                     ;   e_ident
              db      1, 1, 1, 0, 0
_start:       mov     bl, 42
              xor     eax, eax
              inc     eax
              int     0x80
              dw      2                               ;   e_type
              dw      3                               ;   e_machine
              dd      1                               ;   e_version
              dd      _start                          ;   e_entry
              dd      phdr - $$                       ;   e_phoff
              dd      0                               ;   e_shoff
              dd      0                               ;   e_flags
              dw      ehdrsize                        ;   e_ehsize
              dw      phdrsize                        ;   e_phentsize
              dw      1                               ;   e_phnum
              dw      0                               ;   e_shentsize
              dw      0                               ;   e_shnum
              dw      0                               ;   e_shstrndx

ehdrsize      equu    $ - ehdr

phdr:                                                 ; Elf32_Phdr
              dd      1                               ;   p_type
              dd      0                               ;   p_offset
              dd      $$                              ;   p_vaddr
              dd      $$                              ;   p_paddr
              dd      filesize                        ;   p_filesz
              dd      filesize                        ;   p_memsz
              dd      5                               ;   p_flags
              dd      0x1000                          ;   p_align

phdrsize      equ     $ - phdr

filesize      equ     $ - $$
```

After all, bytes are bytes!

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
$ wc -c a.out
     84 a.out
```

Not bad, eh?

Now we've really gone as low as we can go. Our file is exactly as long as one ELF header and one program header table entry, both of which we absolutely require in order to get loaded into memory and run. So there's nothing left to reduce now!

Except ...

Well, what if we could do the same thing to the program header table that we just did to the program? Have it overlap with the ELF header, that is. Is it possible?

It is indeed. Take a look at our program. Note that the last eight bytes in the ELF header bear a certain kind of resemblence to the first eight bytes in the program header table. A certain kind of resemblence that might be described as "identical".

So ...

```
; tiny.asm

BITS 32

              org     0x08048000

ehdr:
              db      0x7F, "ELF"             ; e_ident
              db      1, 1, 1, 0, 0
_start:       mov     bl, 42
              xor     eax, eax
              inc     eax
              int     0x80
              dw      2                       ; e_type
              dw      3                       ; e_machine
              dd      1                       ; e_version
              dd      _start                  ; e_entry
              dd      phdr - $$               ; e_phoff
              dd      0                       ; e_shoff
              dd      0                       ; e_flags
              dw      ehdrsize                ; e_ehsize
              dw      phdrsize                ; e_phentsize
phdr:         dd      1                       ; e_phnum       ; p_type
                                              ; e_shentsize
              dd      0                       ; e_shnum       ; p_offset
                                              ; e_shstrndx
ehdrsize      equ     $ - ehdr
              dd      $$                                      ; p_vaddr
              dd      $$                                      ; p_paddr
              dd      filesize                                ; p_filesz
              dd      filesize                                ; p_memsz
              dd      5                                       ; p_flags
              dd      0x1000                                  ; p_align
phdrsize      equ     $ - phdr

filesize      equ     $ - $$
```

And sure enough, Linux doesn't mind our parsimony one bit:

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
$ wc -c a.out
     76 a.out
```

Now we've *really* gone as low as we can go. There's no way to overlap the two structures any more than this. The bytes simply don't match up. This is the end of the line!

Unless, that is, we could change the contents of the structures to make them match even further....

How many of these fields is Linux actually looking at, anyway? For example, does Linux actually check to see if the e_machine field contains 3 (indicating an Intel 386 target), or is it just assuming that it does?

As a matter of fact, in that case it does. But a surprising number of other fields are being quietly ignored.

So: Here's what is and isn't essential in the ELF header. The first four bytes have to contain the magic number, or else Linux won't touch it. The other three bytes in the e_ident field are not checked, however, which means we have no less than twelve contiguous bytes we can set to anything at all. e_type has to be set to 2, to indicate an executable, and e_machine has to be 3, as just noted. e_version is, like the version number inside e_ident, completely ignored. (Which is sort of understandable, seeing as currently there's only one version of the ELF standard.) e_entry naturally has to be valid, since it points to the start of the program. And clearly, e_phoff needs to contain the correct offset of the program header table in the file, and e_phnum needs to contain the right number of entries in said table. e_flags, however, is documented as being currently unused for Intel, so it should be free for us to reuse. e_ehsize is supposed to be used to verify that the ELF header has the expected size, but Linux pays it no mind. e_phentsize is likewise for validating the size of the program header table entries. This one was unchecked in older kernels, but now it needs to be set correctly. Everything else in the ELF header is about the section header table, which doesn't come into play with executable files.

And now how about the program header table entry? Well, p_type has to contain 1, to mark it as a loadable segment. p_offset really needs to have the correct file offset to start loading. Likewise, p_vaddr needs to contain the proper load address. Note, however, that we're not required to load at 0x08048000. Almost any address can be used as long as it's above 0x00000000, below 0x80000000, and page-aligned. The p_paddr field is documented as being ignored, so that's guaranteed to be free. p_filesz indicates how many bytes to load out of the file into memory, and p_memsz indicates how large the memory segment needs to be, so these numbers ought to be relatively sane. p_flags indicates what permissions to give the memory segment. It needs to be readable (4), or it won't be usable at all, and it needs to also be executable (1), or else we can't execute code in it. Other bits can probably be set as well, but we need to have those at minimum. Finally, p_align gives the alignment requirements for the memory segment. This field is mainly used when relocating segments containing position-independent code (as for shared libraries), so for an executable file Linux will ignore whatever garbage we store here.

All in all, that's a fair bit of leeway. In particular, a bit of scrutiny will reveal that most of the necessary fields in the ELF header are in the first half - the second half is almost completely free for munging. With this in mind, we can interpose the two structures quite a bit more than we did previously:

```
; tiny.asm

BITS 32

              org     0x00200000

              db      0x7F, "ELF"             ; e_ident
              db      1, 1, 1, 0, 0
```

```
_start:
              mov     bl, 42
              xor     eax, eax
              inc     eax
              int     0x80
              dw      2                       ; e_type
              dw      3                       ; e_machine
              dd      1                       ; e_version
              dd      _start                  ; e_entry
              dd      phdr - $$               ; e_phoff
phdr:         dd      1                       ; e_shoff       ; p_type
              dd      0                       ; e_flags       ; p_offset
              dd      $$                      ; e_ehsize      ; p_vaddr
                                              ; e_phentsize
              dw      1                       ; e_phnum       ; p_paddr
              dw      0                       ; e_shentsize
              dd      filesize                ; e_shnum       ; p_filesz
                                              ; e_shstrndx
              dd      filesize                                ; p_memsz
              dd      5                                       ; p_flags
              dd      0x1000                                  ; p_align

filesize      equ     $ - $$
```

As you can (hopefully) see, the first twenty bytes of the program header table now overlap the last twenty bytes of the ELF header. The two dovetail quite nicely, actually. There are only two parts of the ELF header within the overlapped region that matter. The first is the e_phnum field, which just happens to coincide with the p_paddr field, one of the few fields in the program header table which is definitely ignored. The other is the e_phentsize field, which coincides with the top half of the p_vaddr field. These are made to match up by selecting a non-standard load address for our program, with a top half equal to 0x0020.

Now we have *really* left behind all pretenses of portability ...

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
$ wc -c a.out
     64 a.out
```

... but it works! And the program is twelve bytes shorter, exactly as predicted.

This is where I say that we can't do any better than this, but of course, we already know that we can — if we could get the program header table to reside *completely* within the ELF header. Can this holy grail be achieved?

Well, we can't just move it up another twelve bytes without hitting hopeless obstacles trying to reconcile several fields in both structures. The only other possibility would be to have it start immediately following the first four bytes. This puts the first part of the program header table comfortably within the e_ident area, but still leaves problems with the rest of it. After some experimenting, it looks like it isn't going to quite be possible.

However, it turns out that there are still a couple more fields in the program header table that we can pervert.

We noted that p_memsz indicates how much memory to allocate for the memory segment. Obviously it needs to be at least as big as p_filesz, but there wouldn't be any harm if it was larger. Just because we ask for memory doesn't mean we have to use it, after all.

Secondly, it turns out that, contrary to all my expectations, the executable bit can be dropped from the p_flags field. It turns out that the readable and executable bits are redundant: either one will imply the other.

So, with these facts in mind, we can reorganize the file into this little monstrosity:

```
; tiny.asm

BITS 32

              org     0x00010000

              db      0x7F, "ELF"             ; e_ident
              dd      1                                       ; p_type
              dd      0                                       ; p_offset
              dd      $$                                      ; p_vaddr
              dw      2                       ; e_type        ; p_paddr
              dw      3                       ; e_machine
              dd      _start                  ; e_version     ; p_filesz
              dd      _start                  ; e_entry       ; p_memsz
              dd      4                       ; e_phoff       ; p_flags
_start:
              mov     bl, 42                  ; e_shoff       ; p_align
              xor     eax, eax
              inc     eax                     ; e_flags
              int     0x80
              db      0
              dw      0x34                    ; e_ehsize
              dw      0x20                    ; e_phentsize
              dw      1                       ; e_phnum
              dw      0                       ; e_shentsize
              dw      0                       ; e_shnum
              dw      0                       ; e_shstrndx

filesize      equ     $ - $$
```

The p_flags field has been changed from 5 to 4, as we noted we could get away with doing. This 4 is also the value of the e_phoff field, which gives the offset into the file for the program header table, which is exactly where we've located it. The program (remember that?) has been moved down to lower part of the ELF header, beginning at the e_shoff field and ending inside the e_flags field.

Note that the load address has been changed to a much lower number — about as low as it can be, in fact. This keeps the value in the e_entry field to a reasonably small number, which is good since it's also the p_memsz number. (Actually, with virtual memory it hardly matters — we could have left it at our original value and it would work just as well. But there's no harm in being polite.)

The change to p_filesz may require an explanation. Because we aren't setting the write bit in the p_flags field, Linux won't let us define a p_memsz value greater than p_filesz, since it can't zero-initialize those extra bytes if they aren't writeable. Since we can't change the p_flags field without moving the program header table out of alignment, you might think that the only solution would be to lower the p_memsz value back down to equal p_filesz (which would make it impossible to share it with e_entry). However, another solution exists, namely to increase p_filesz to equal p_memsz. That means they're both larger than the real file size — quite a bit larger, in fact — but it absolves the loader from having to write to read-only memory, which is all it cared about.

And so ...

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
$ wc -c a.out
     52 a.out
```

... and so, with both the program header table and the program itself completely embedded within the ELF header, our executable file is now exactly as big as the ELF header! No more, no less. And *still* running without a single complaint from Linux!

Now, finally, we have truly and certainly reached the absolute minimum possible. There can be no question about it, right? After all, we have to have a complete ELF header (even if it is badly mangled), or else Linux wouldn't give us the time of day!

Right?

Wrong. We have one last dirty trick left.

It seems to be the case that if the file isn't quite the size of a full ELF header, Linux will still play ball, and fill out the missing bytes with zeros. We have no less than seven zeros at the end of our file, and if we drop them from the file image:

```
; tiny.asm

BITS 32

              org     0x00010000

              db      0x7F, "ELF"             ; e_ident
              dd      1                                       ; p_type
              dd      0                                       ; p_offset
              dd      $$                                      ; p_vaddr
              dw      2                       ; e_type        ; p_paddr
              dw      3                       ; e_machine
              dd      _start                  ; e_version     ; p_filesz
              dd      _start                  ; e_entry       ; p_memsz
              dd      4                       ; e_phoff       ; p_flags
_start:
              mov     bl, 42                  ; e_shoff       ; p_align
              xor     eax, eax
              inc     eax                     ; e_flags
              int     0x80
              db      0
              dw      0x34                    ; e_ehsize
              dw      0x20                    ; e_phentsize
              db      1                       ; e_phnum
                                              ; e_shentsize
                                              ; e_shnum
                                              ; e_shstrndx
```

```
filesize      equ     $ - $$
```

... we can, incredibly enough, still produce a working executable:

```
$ nasm -f bin -o a.out tiny.asm
$ chmod +x a.out
$ ./a.out ; echo $?
42
$ wc -c a.out
     45 a.out
```

Here, at last, we have honestly gone as far as we can go. There is no getting around the fact that the 45th byte in the file, which specifies the number of entries in the program header table, needs to be non-zero, needs to be present, and needs to be in the 45th position from the start of the ELF header. We are forced to conclude that there is nothing more that can be done.

This forty-five-byte file is less than one-eighth the size of the smallest ELF executable we could create using the standard tools, and is less than one-fiftieth the size of the smallest file we could create using pure C code. We have stripped everything out of the file that we could, and put to dual purpose most of what we couldn't.

Of course, half of the values in this file violate some part of the ELF standard, and it's a wonder that Linux will even consent to sneeze on it, much less give it a process ID. This is not the sort of program to which one would normally be willing to confess authorship.

On the other hand, every single byte in this executable file can be accounted for and justified. How many executables have you created lately that you can say *that* about?

(next)
