The ELF Object File Format by Dissection

Software

by Eric Youngdale on May 1, 1995

Last month, we reached a point where were beginning to dissect some real ELF files. For this, I will use the **readelf** utility which I wrote when I was first trying to understand the ELF format itself. Later on, it became a valuable tool for debugging the linker as I added support for ELF. The sources to readelf should be on tsx-11.mit.edu in pub/linux/packages/GCC/src or in pub/linux/BETA/bcc2.

Let us start with a very simple program—the hello world program we used last month.

```
largoh cat hello.c
main()
{
        printf("Hello World\n");
}
largoh gcc-elf -c hello.c
```

On my laptop, the gcc-elf command invokes the ELF version of gcc—once ELF becomes the default format, you will be able to use the regular gcc command which produces the ELF file hello.o. Each ELF file starts with a header (**struct elfhdr** in /usr/include/linux/elf.h), and the readelf utility can display the contents of all of the fields:

```
largoh readelf -h hello.o
ELF magic:7f 45 4c 46 01 01 01 00 00 00 00 00
          00 00 00 00
Type, machine, version = 1 3 1
Entry, phoff, shoff, flags = 0 0 440 0
ehsize, phentsize, phnum = 52 0 0
shentsize, shnum, shstrndx = 40 11 8
```

The ELF **magic** field is just a way of unambiguously identifying this as an ELF file. If a file does not contain those 16 bytes in the **magic** field, it is not an ELF file. The type, machine, and version fields identify this as an ET_REL file (i.e., an object file) for the i386. The **ehsize** field is just the **sizeof(struct elfhdr)**.

Each ELF file contains a table that describes the sections within the file. The **shnum** field indicates that there are 11 sections; the **shoff** field indicates that the section header table starts at byte offset 440 within the file. The **shentsize** field indicates that the entry for each section is 40 bytes long. All throughout ELF, the sizes of various structures are always explicitly stated. This allows for flexibility; the structures can be expanded as required for some hardware platforms and the standard ELF tools do not have to know about this to be able to make sense of the binary. Also, it allows room for future expansion of the structures by newer versions of the standard.

```
largoh readelf -S hello.o
There are 11 section headers, starting at offset 1b8:
[0]  NULL            00000000 00000 00000 00 / 0 0 0 0
[1] .text            PROGBITS     00000000 00040 00014 00 / 6 0 0 10
[2] .rel.text        REL          00000000 00370 00010 08 / 0 9 1 4
[3] .data            PROGBITS     00000000 00054 00000 00 / 3 0 0 4
[4] .bss             NOBITS       00000000 00054 00000 00 / 3 0 0 4
[5] .note            NOTE         00000000 00054 00014 00 / 0 0 0 1
[6] .rodata          PROGBITS     00000000 00068 00004 00 / 2 0 0 1
[7] .comment         PROGBITS     00000000 00075 00012 00 / 0 0 0 1
[8] .shstrtab        STRTAB       00000000 00087 00042 00 / 0 0 0 1
[9] .symtab          SYMTAB       00000000 00364 000c0 10 / 0 a a 4
[a] .strtab          STRTAB       00000000 00194 00024 00 / 0 0 0 1
```

Listing 1. Section Table for hello.o

Each section header is just a **struct ELF32_Shdr**. You may notice that the name field is just a number—this is not a pointer, but an offset into the .shstrtab section (we can find the index of the .shstrtab section from the file header in the **shstrndx** field). Thus we find the name of each section at the specified offset within the .shstrtab section. Let us dump the section table for this file; see figure 1. You will notice sections for nearly everything which we have already discussed. Each section has an identifier which specifies what the section contains (in general, you should never have to actually know the name of a section or compare it to anything).

After the type, there is a series of numbers. The first of these is the address in virtual memory where this section should be loaded. Since this is a .o file, it is not intended to be loaded into virtual memory, and this field is not filled in. Next is the offset within the file of the section, and then is the size of the section. After this come a series of numbers—I won't parse these in detail for you, but they contain things like the required alignment of the section, a set of flags which indicate whether the section is read-only, writable, and/or executable.

The readelf program is capable of performing disassembly:

```
largoh readelf -i 1 hello.o
0x00000000  pushl    %ebp
0x00000001  movl     %esp,%ebp
0x00000003  pushl    $0x0
0x00000008  call     0x00007559
0x0000000d  addl     $4,%esp
0x00000010  movl     %ebp,%esp
0x00000012  popl     %ebp
0x00000013  ret
```

The .rel.text section contains the relocations for the .text section of the file, and we can display them as follows:

```
largoh readelf -r hello.o
Relocation section data:.rel.text (0x2 entries)
Tag: 00004 Value 00301 R_386_32   (0 )
Tag: 00009 Value 00d02 R_386_PC32  (0 printf)
```

This indicates that the .text section has two relocations. As expected, there is a relocation for printf indicating that we must patch the address of printf into offset 9 from the beginning of the .text section, which happens to be the operand of the call instruction. There is also a relocation so that we pass the correct address to printf.

Now let us see what happens when this file is linked into an executable. The section table now looks something like Listing 2.

The first thing you will notice is a lot more sections than were in the simple .o file. Much of this because this file requires the ELF shared library libc.so.1.

At this point I should mention the mechanics of what happens when you run an ELF program. The kernel looks through the binary and loads it into the user's virtual memory. If the application is linked to a shared library, the application will also contain the name of the dynamic linker that should be used. The kernel then transfers control to the dynamic linker, not to the application. The dynamic loader is responsible for first initializing itself, loading the shared libraries into memory, resolving all remaining relocations, and then transferring control to the application.

Going back to our executable, the .interp section simply contains an ASCII string that is the name of the dynamic loader. Currently this will always be /lib/ld10-linux.so.1 (the dynamic loader itself is also an ELF shared library).

Next you will notice 3 sections, called .hash, .dynsym, and .dynstr. This is a minimal symbol table used by the dynamic linker when performing relocations. You will notice that these sections are mapped into virtual memory (the virtual address field is non-zero). At the very end of the image are the regular symbol and string tables, and these are not mapped into virtual memory by the loader. The .hash section is just a hash table that is used so that we can quickly locate a given symbol in the .dynsym section, thereby avoiding a linear search of the symbol table. A given symbol can typically be located in one or two tries through the use of the hash table.

The next section I want to mention is the .plt section. This contains the jump table that is used when we call functions in the shared library. By default the .plt entries are all initialized by the linker not to point to the correct target functions, but instead to point to the dynamic loader itself. Thus, the first time you call any given function, the dynamic loader looks up the function and fixes the target of the .plt so that the next time this .plt slot is used we call the correct function. After making this change, the dynamic loader calls the function itself.

This feature is known as lazy symbol binding. The idea is that if you have lots of shared libraries, it could take the dynamic loader lots of time to look up all of the functions to initialize all of the .plt slots, so it would be preferable to defer binding addresses to the functions until we actually need them. This turns out to be a big win if you only end up using a small fraction of the functions in a shared library. It is possible to instruct the dynamic loader to bind addresses to all of the .plt slots before transferring control to the application—this is done by setting the environment variable **LD_BIND_NOW=1** before running the program. This turns out to be useful in some cases when you are debugging a program, for example. Also, I should point out that the .plt is in read-only memory. Thus the addresses used for the target of the jump are actually stored in the .got section. The .got also contains a set of pointers for all of the global variables that are used within a program that come from a shared library.

The .dynamic section contains some shorthand notes used by the dynamic loader. You will notice that the section table is not itself loaded into virtual memory, and in fact it would not be good for performance for the dynamic loader to have to try to parse it to figure out what needs to be done. The .dynamic section is essentially just a distilled version of the section header table that contains just what is needed for the dynamic loader to do its job.

You will notice that since the section header table is not loaded into memory, neither the kernel nor the dynamic loader will be able to use that table when loading files into memory. A shorthand table of program headers is added to provide a distilled version of the section table containing just the information required to load a file into memory. For the above file it looks something like:

```
largoh readelf -l hello
Elf file is Executable
Entry point 0x8000400
There are 5 program headers, starting at offset 34:
PHDR          0x00034 0x08000034 0x0000a0 0x000a0 R E
Interp        0x000d4 0x080000d4 0x00017 0x00017 R
Requesting program interpreter \
[/lib/elf/ld-linux.so.1]
Load          0x00000 0x08000000 0x00515 0x00515 R E
Load          0x00518 0x08001518 0x000cc 0x00004 RW
Dynamic       0x00534c 0x0800154c 0x00098 0x00098 RW
Shared library: [libc.so.4] 1
```

As you can see, the program header contains a pointer to the name of the dynamic loader, instructions on what portions of the file are to be loaded into virtual memory (and the virtual addresses they should be loaded to), the permissions of the segments of memory, and finally a pointer to the .dynamic section that the dynamic loader will need. Note that the list of required shared libraries is stored in the .dynamic section.

I will not pick apart an ELF shared library for you here—libraries look quite similar to ELF executables. If you are interested, you can get the readelf utility and pick apart your own libraries.

At the start of this article, I said one reason we were switching to ELF was that it was easier to build shared libraries with ELF. I will now demonstrate how. Consider two files:

```
largoh cat hello1.c
main()
{
        greet();
}
largoh cat english.c
greet()
{
        printf("Hello World\n");
}
```

The idea is that we want to build a shared library from english.c, and link hello1 against it. The commands to generate the shared library are:

```
largoh gcc-elf -fPIC -c english.c
largoh gcc-elf -shared -o libenglish.so english.o
```

That's all there is to it. Now we compile and link the hello1 program:

```
largoh gcc-elf -c hello1.c
largoh gcc-elf -o hello1 hello1.o -L. -lenglish
```

And finally we can run the program. Normally the dynamic loader only looks in certain locations for shared libraries, and the current directory is not one of the places it normally looks. Thus to run the program, you can use a command like:

```
largoh LD_LIBRARY_PATH=. ./hello1
Hello World
```

The environment variable **LD_LIBRARY_PATH** tells the dynamic loader to look in additional places for the shared libraries (this feature is disabled for setuid programs for security reasons).

To avoid having to specify **LD_LIBRARY_PATH**, you have several options. You could copy your shared library to /lib/elf, but you can also link your program in the following way:

```
largoh gcc-elf -o hello1 hello1.o\ /home/joe/libenglish.so
largoh ./hello1
Hello World
```

To build more complicated shared libraries, the procedure is not really that much different. Everything that you want to put into the shared library should be compiled with **-fPIC**, when you have compiled everything, you just link it all together with the **gcc -shared** command.

The procedure is so much simpler mainly because we bind addresses to functions at runtime. With a out libraries, the addresses are bound at link time. This means that lots of special care must be taken to ensure that the .plt and .got have sufficient room for future expansion and that we keep the variables at the same addresses from one version of the library to the next. The tools for building a.out libraries help ensure all of this, but it makes the build procedure much more complicated.

ELF offers one further feature that is not easily available with a.out. The **dlopen()** function can be used to dynamically load a shared library into the user's memory, and you are then able to call the dynamic loader to find symbols within this shared library—in other words, you can call functions that are defined in these modules. In addition, the dynamic loader is used to resolve any undefined symbols within the module itself.

This may be easiest to explain with an example. Given the following source file:

```
#include <dlfcn.h>
main(int argc, char * argv[])
{
  void (*greeting)();
  void * module;
  if( argc > 2 ) exit(0);
  module = dlopen(argv[1], RTLD_LAZY);
 if(!module) exit(0);
  greeting = dlsym(module, "greet");
  if(greeting) {
    (*greeting)();
  }
  dlclose(module);
 }
```

you can compile, link, and run it (using the shared library english.so which was built earlier):

```
largoh gcc-elf -c hello2.c
largoh gcc-elf -o hello2 hello2.o -ldl
largoh ./hello2 ./libenglish.so
Hello World
```

To expand this example a little bit, you could generate other modules with greetings in other languages. Thus in theory, one could add multi-lingual support for some application merely by supplying a set of shared libraries that contain the language-specific portions of the application. In the above example, I showed how you can locate the address of a function within a shared library. But the **dlsym()** function will also return the address of data variables, so you could just as easily retrieve the address of a text string from the shared library.

As I prepare to close, I should mention some options to readelf which I have not demonstrated. **readelf -s** dumps the symbol tables and **readelf -d** dumps the .dynamic section.

Finally, I should mention something about the timetable. When we first got ELF to a point where it was usable (last September), we decided to spend a relatively long period of time testing it and shaking out all of the problems. Back then I felt that roughly 4-to-6 months would allow people to test it thoroughly, plus we wanted to give an opportunity for certain applications to be adapted for ELF (the most recent versions of inmod and Wine now support ELF, for example). As I write this, no firm date has been set for a public release, but it is possible that ELF will be public by the time you read this.

In these articles I have attempted to give you a guided introduction to the ELF file format. A lot of the material I have covered is not of much practical value to most users (unless you want to hack the linker), but my experience is that there are a lot of people who are curious about how it all works, and I hope that I have provided enough information to satisfy most people.

For more information about the ELF file format, you can obtain the ELF specifications from a number of sources—you can try ftp.intel.com in pub/tis/elf11.zip. The specifications are also available in a printed format. See *SYSTEM V Application Binary Interface* (ISBN 0-13-100439-5) and *SYSTEM V Application Binary Interface, Intel386 Architecture Processor Supplement* (ISBN 0-13-104070-5).

Eric Youngdale has worked with Linux for over three years, and has been active in kernel development. He developed the current a.out Linux shared libraries before developing much of the new ELF support. He can be reached as eric@ajb.com.