Understanding the ELF

James Fisher Follow
Jan 25, 2015 · 15 min read

I wrote an ELF binary which, when run, prints itself on standard output:

```
$ ./quine > quine2
$ md5sum quine quine2
5d24b05850f451b5b1e1a87a7c5d57c3  quine
5d24b05850f451b5b1e1a87a7c5d57c3  quine2
```

It doesn't "cheat" by reading its own file. Instead, it uses how ELF binary files are loaded into memory. To explain how the program works, I'll show you it byte-by-byte. Along the way, we'll learn about files, programs, the ELF format, and tools for working with these things.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Files and hexadecimal

The program *./quine* is just a file. This means it is a named list of bytes, just like text files, images, videos, etc. We can view the bytes of any file using a program called *hexdump*. Here's how we can use it for our program:

```
$ hexdump ./quine
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0002 003e 0001 0000 0078 0040 0000 0000
0000020 0040 0000 0000 0000 0000 0000 0000 0000
0000030 0000 0000 0040 0038 0001 0000 0000 0000
0000040 0001 0000 0005 0000 0000 0000 0000 0000
0000050 0000 0040 0000 0000 0000 0000 0000 0000
0000060 0095 0000 0000 0000 0095 0000 0000 0000
0000070 0000 0020 0000 0000 95ba 0000 b900 0000
0000080 0040 01bb 0000 b800 0004 0000 80cd 01b8
0000090 0000 cd00 0080
0000095
```

What does this output mean? The first column is just a counter; ignore it. The other columns express the bytes of the file in hexadecimal notation ("hex").

When talking about bytes, hex is the universal language. In hex, we write numbers using the 16 characters 0..9 and a..f. This is just like decimal with

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

six extra characters, making it base-16 instead of base-10. A byte can be written using two hex characters. For example, the first byte of this file is ‘7f’, the second byte is ‘45’, and the last byte is ‘80’. Since these numbers can be confused with decimal, we prefix them with ‘0x’. We say the first byte of the file is 0x45 so we don’t confuse it with decimal 45. By decimal conversion, we can write that 0x45 = 4*16 + 5 = 69 in decimal.

Personally, I find the default hexdump view confusing. Let’s make it print the file without the “sidebar”, and with every byte separated by spaces:

```
$ hexdump -e '16/1 "%02x " "\n"' -v quine
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
02 00 3e 00 01 00 00 00 78 00 40 00 00 00 00 00
40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 40 00 38 00 01 00 00 00 00 00 00 00
01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00
00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00
95 00 00 00 00 00 00 00 95 00 00 00 00 00 00 00
00 00 20 00 00 00 00 00 ba 95 00 00 00 b9 00 00
40 00 bb 01 00 00 00 b8 04 00 00 00 cd 80 b8 01
00 00 00 cd 80
```

The ELF header

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

So what do all those bytes *mean*? On Linux, executable programs are written in a format called ELF ("Executable and Linkable Format"). Files in this format tell Linux how to create a new process. When we 'execute' a file, Linux reads the file, uses the information in it to set up a new process, then sets that process running. This process is known as "loading", and the part of Linux that converts an ELF file into a process is called the "loader".

The first 64 bytes of the ELF file are the 'ELF header'. We can view just the ELF header by asking hexdump to just print the first 64 bytes:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 64 quine
7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
02 00 3e 00 01 00 00 00 78 00 40 00 00 00 00 00
40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 40 00 38 00 01 00 00 00 00 00 00 00
```

The first four bytes of an ELF header are always the same; they are a "magic number" which says "hey, I'm an ELF file". (The second, third, and fourth bytes [45 4c 46] are the string "ELF" encoded in ASCII.)

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The next three bytes tell us how to interpret the rest of the file. Byte 5 tells us whether we're using a the 32-bit or 64-bit ELF format. In our file, byte 5 has the value 2, which means "64-bit". We can read individual bytes from the file using the *hexdump -s* flag in combination with *-n*:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 1 -s 4 quine
02
```

Byte 6 tells us the 'data encoding' in the rest of the file. The data encoding is a way to encode numbers as lists of bytes. For our file, this has value 1, which tells us that the file uses "least-significant-byte", or "LSB", encoding. I'll explain how to read LSB encoding soon, when we get to a number encoded in it.

Byte 7 tells us which version of the ELF format the file is using. All files use ELF version 1, because that's the only version which exists!

Bytes 8 through 16 are zeroed out. They are just "buffer" space with no meaning. Why, you ask? A couple of reasons. First, the buffer provides

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

extensibility, so that future ELF versions can add more fields at this location. Second, the buffer pushes the next few fields into “alignment”. I’ll explain this “alignment” concept soon, too.

Bytes 17 and 18 are a two-byte field. Our file has the value:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 2 -s 16 quine
02 00
```

Now the “data encoding” comes into play. Our ELF file has encoded a number as the bytes [02 00] using LSB encoding. If our file was using “most-significant-byte” data encoding, we would read these two bytes by concatenating the characters as the number 0x0200 = 512 in decimal. However, our file uses least-significant-byte (LSB) encoding, which means we reverse the order of the bytes before reading them as hex: [02 00] becomes [00 02]. Note that we reverse the order of the *bytes*, not the *characters*: [02 00] does *not* get reversed to [00 20]! So our two-byte field above is read as 0x0002 = 0x2 = 2.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

This field is the ‘type’ of our ELF file. The value 2 means that our file is an ‘executable’. Other ELF file types include libraries (object files which end in .o), but we won’t get into that.

The next field is another two-byte field called ‘machine’. In our file this is [3e 00] = 0x003e = 0x3e = 3*16 + 14 = 62. The value 62 means “x86–64”. This means we expect this executable to run on x86–64 machines, and not necessarily any other architectures.

The next field is a four-byte field called “version”, which, confusingly, tells us the ELF version again. It’s always set to 1. Before looking, take a guess at what the number 1 looks like in a four-byte LSB field. Then use hexdump to check:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 4 -s 20 quine
01 00 00 00
```

Now about that “alignment” thing. Notice that this four-byte field starts at byte number 20. The number 20 is a multiple of four. All four-byte fields

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

must start at an index which is a multiple of four. So this field could not have started at byte 21, or 22, or 23. More generally, all *n*-byte fields must start at an index which is a multiple of *n*. This is a rule of the ELF format. You will notice that the previous 2-byte fields all started at multiples of 2. When a field obeys this rule, we say that the field is “aligned”.

The next field is an 8-byte field. It starts at index 24, which is — yep — a multiple of 8. In our file it looks like this:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 8 -s 24 quine
78 00 40 00 00 00 00 00
```

Remember that the ELF file instructs Linux how to set up a process and start it running? This field says where to start running. It is the address at which execution will start when the process begins. The field is 8 bytes, or 64 bits, because it is a memory address for a 64-bit platform. For plain old x86, this field is 4 bytes (32 bits). Our program starts at address *4194424*. If that strikes you as kinda random, don’t worry; we’ll get to it.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The next few fields deal mainly with “program segments” and “sections”. Sections contain metadata useful for linking, debugging, and other things, but are not required for actually running the program. Since we are only interested in running the program, rather than linking or debugging it, our ELF file contains no sections, and we won’t cover anything about them here.

Program segments, on the other hand, are important. They have multiple uses, but our program only uses them in one way: to specify data to load into memory before starting the process.

So, the next field is the “program header table offset”, and it’s another 8-byte field, at index 32:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 8 -s 32 quine
40 00 00 00 00 00 00 00
```

This value is 0x40 = 64. The program header table offset tells us where in the file we will find the *program header table*. Our previous fields all had fixed positions in the file, but the program header table is different — its

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

location is indicated by a *pointer* to it. This field is that pointer, and the value 64 tells us that the program header table starts at index 64.

These "pointer" or "offset" fields may be unfamiliar to you if you're used to formats like XML or JSON, which can be defined by context-free grammars. You should think of these offset fields as analogous to pointers in C. They are used many times in the ELF format, and are necessary to how the quine works.

The next field is the "section header table offset". Our file has no section header table, since it has no sections. This field is arbitrarily set to 0.

The next field, the "ELF header size", is also set to 64. This simply tells us that the ELF header that we're currently reading will end at index 64. (This happens to also be where the program header table starts: the ELF header is immediately followed by the program header table.)

The next two fields in the ELF header tell us how to read the program header table. The "program header entry size" tells us how many bytes long a single entry in the program header table is. The next field, the "number of

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

program headers", is self-explanatory. For our file, each program header is 56 bytes, and there is only 1 program header.

The program header table simply consists of program headers laid out consecutively, like an array in C. Since each header is 56 bytes, and we only have one header, we know the entire program header table is 56 bytes.

The final three fields in the ELF header are two-byte fields that relate to sections. We're not interested in sections here, and our program doesn't have any. These three fields are zeroed out.

The program header table

We're finally out of the ELF header, and into the program header table! We learned that the program header table starts at index 64, and runs for 56 bytes. So we can now print the program header table:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 56 -s 64 quine
01 00 00 00 05 00 00 00 00 00 00 00 00 00 00 00
00 00 40 00 00 00 00 00 00 00 40 00 00 00 00 00
95 00 00 00 00 00 00 00 95 00 00 00 00 00 00 00
00 00 20 00 00 00 00 00
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

We also learned that there is only one header in the table, and it runs the entire length of the table. So the above is not only the program header table, it's also the single program header.

Let's decode it. Just as with the ELF header, the program header consists of multi-byte fields. There are two four-byte fields, then the rest are 8-byte fields. Notice that this means that the table must be aligned to an 8-byte location, which indeed it is.

The first field, "program header type", is set to 1. This "indicates that this program header describes a segment to be loaded from the file." The next fields are going to reference another part of the file, the segment itself. That segment is to be loaded into a specified place in memory when the process is set up.

The second field, "flags", is set to 5 when decoded as a decimal. However, we shouldn't view this field as a decimal; we should view it as a "bitfield". A bitfield encodes a set of boolean values, in the following way. The value 5 when seen as a sum of unique powers of 2 is equal to 4 + 1, which means that the flags corresponding to the numbers 4 and 1 are considered "on".

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The number 4 is the flag called “read”, and the number 1 is the flag called “execute”. The only other possible flag is one called “write”, which is coded as the number 2, which is not part of 4 + 1, and so is considered “off”.

So our program segment has “read” and “execute” flags turned on, but the “write” flag turned off. These flags tell Linux what the running process is allowed to do with the memory that we’re going to load this segment into. The “read” flag says that the process can read that memory (e.g. load it into registers). The “execute” flag says that the memory can be decoded and executed as instructions.

If the segment had had the “write” flag, it would have been able to write to that memory too. Because it does not have that permission, attempts to write to those memory locations will result in a “segmentation fault.” You may have heard this term before — notice that it’s talking about the *segments* that our ELF file is defining!

This read/execute permission combination is typically used for segments containing the program. They usually don’t have the write permission, because it’s unconventional for programs to self-modify.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

(Confusingly, some guides say that “flags” is not the second field of the header, but the second-to-last. This doesn’t gel with my experience.)

The next field is an eight-byte field called “offset”. It is the index into the ELF file at which our segment starts. In our file it is set to 0. Yes, index 0 is the start of the ELF file, starting at that magic number! This is key to how the program prints itself: we are going to load the entire ELF file into a specific location in memory, so that the program can print it out.

It may seem odd to you that the program segment in the file can overlap the ELF header and other parts of the file. This is possible because the content of an ELF file is defined by internal pointers, instead of being “structural” like a typical syntax. It might be unusual for a generated ELF file to exploit the possibility of “overlap”, but it’s not forbidden.

The next two fields in the program header are eight-byte fields called “virtual address” and “physical address”. For Linux application programs like ours, the physical address field is ignored, and only the virtual address is relevant. This is because application programs use an API called *virtual memory*. The virtual memory API exposes 2^64 bytes of memory to our

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

program. The “virtual address” field tells the loader where in that memory to put the start of this segment.

For our program, the virtual address field is set to 2^22, or 4194304. Remember that the “entry point” for our program is set to 4194424. This means the program entry point is byte 120 of this segment.

Next up are another two-byte fields called “file size” and “memory size”. The “file size” field tells us how many bytes long the segment is in the file. The “memory size” field is usually the same, but can be larger (in which case the trailing memory is zeroed out). For our file, both fields are set to 149. This is the size of the entire file; hence we copy the entire file into memory as a single segment, starting at address 4194304 and ending at address 4194453.

The final field of the row is an eight-byte field called “align”. There are two rules associated with this field. First, it must be an integer which is a power of 2. Second, *virtual address* ≡ *offset*, modulo *align*. (Or rather, *virtual address % align = offset % align*.) Since our offset is 0, and $0 \% x = x$, we can set this field to any power of 2 that we like. In our file, we happen to set

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

“align” to 0x200000, or 2^21. You could always set it to 1 for simplicity, since that will always be valid.

(That said, I don’t actually understand the *purpose* of the “align” field. I’m told it’s to make memory-mapping the file more efficient, in which case larger values are better. But it would be trivial for the OS to find the largest value which satisfies the two constraints above, making this field redundant …)

The program

Finally, we get to the program segment itself! Our program segment runs from byte 0 to byte 148 of the file — i.e., the whole file. The ELF header and program header table take up the first 120 bytes of this. The remaining 29 bytes are the executable program instructions. Here they are:

```
$ hexdump -e '16/1 "%02x " "\n"' -v -n 29 -s 120 quine
ba 95 00 00 00 b9 00 00 40 00 bb 01 00 00 00 b8
04 00 00 00 cd 80 b8 01 00 00 00 cd 80
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Our file defines the entry point to be memory address 4194424. We load the file into memory starting at address 4194304. This means the entry point is 4194424–4194304 = byte 120 of the file. This is the first byte of the above block.

Technically, the *meaning* of this block of data is not part of the ELF specification. As far as ELF is concerned, this is just a meaningless blob of bytes. The meaning is given to this block of bytes by the x86-64 specification, as implemented by x86–64 processors, which understand these bytes as x86–64 instructions. This block of bytes is isomorphic to an assembly program. We can use tools like *nasm* and *ndisasm* to convert between the two representations. To read this block of bytes as instructions, we can run:

```
$ dd skip=120 bs=1 count=29 if=quine 2> /dev/null | ndisasm -b 64 -
00000000  BA95000000        mov edx,0x95
00000005  B900004000        mov ecx,0x400000
0000000A  BB01000000        mov ebx,0x1
0000000F  B804000000        mov eax,0x4
00000014  CD80              int 0x80
00000016  B801000000        mov eax,0x1
0000001B  CD80              int 0x80
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Okay, but why does [BA 95 00 00 00] decode to the instruction *mov edx,0x95*? Unfortunately, the syntax is really hairy, so I plan to cover the specifics in some future post. The important thing to understand is the assembly representation, which is shown here in "Intel syntax":

```
mov edx,0x95
mov ecx,0x400000
mov ebx,0x1
mov eax,0x4
int 0x80
mov eax,0x1
int 0x80
```

We have seven instructions which will be executed from top to bottom. There are two different operations being performed here: *mov* operations and *int* operations. A *mov* instruction takes the form "mov *dest*, *src*", and specifies that some data represented by *src* is to be copied to the location represented by *dest*. Above, the destination is always a register, like "edx" or "ecx", and the source is always a constant hexadecimal value, like "0x95" or "0x1".

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

An *int* instruction takes the form “int *interrupt*”, where interrupt is a constant value. Above, we only use “int 0x80”, which is understood as a *system call* instruction. Our program consists of two blocks, each of which first moves some values into registers, then performs a system call.

A system call requests that Linux to do something. The value in the register *eax* determines which kind of call is being made. The value 4 identifies the the call *sys_write*, and the value 1 identifies the call *sys_exit*. So our program makes a call to *sys_write*, then a call to *sys_exit*.

System calls have zero or more arguments. These arguments are placed into registers prior to making the system call. The sys_write call takes three arguments. The first, the “file descriptor”, is in register *ebx*. The second, the “buffer pointer”, is in *ecx*. The third, the “count” is in *edx*. So our program makes a call to *sys_write* where the file descriptor is 1, the buffer pointer is 0x400000, and the count is 0x95.

The file descriptor tells Linux where to write to, the buffer pointer tells it where to start reading from in memory, and the count tells it how many bytes to write. So the system call tells Linux to print 0x95 (149) bytes of

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

memory, starting at address 0x400000 (4194304), to file descriptor 1. The file descriptor 1 always refers to standard output.

Now what's in memory between address 4194304 and 4194453? It's the program segment we loaded, which is an exact copy of the entire file! So the ELF loader copies the file into memory, then the program in that file copies the memory to standard output.

Finally, the program makes a system call to *sys_exit*. This call takes one argument, the program exit code, in *ebx*. Our program exits with code 1, but notice that our program doesn't explicitly load anything into *ebx*. Instead, it uses the value previously loaded into *ebx* before calling *sys_write*, which is possible because system calls do not change the state of the registers.

Further reading

"ELF101: a Linux executable walkthrough", Julia Evans.

"Linker and Libraries", Oracle.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

“Executable and Linkable Format”, Sky.

“Linkers and Loaders”, John Levine.

353 claps

WRITTEN BY

James Fisher

I don’t blog here any more; see https://jameshfisher.com. Also see Vidrio for Mac, the holographic presentation app from the future: https://vidr.io

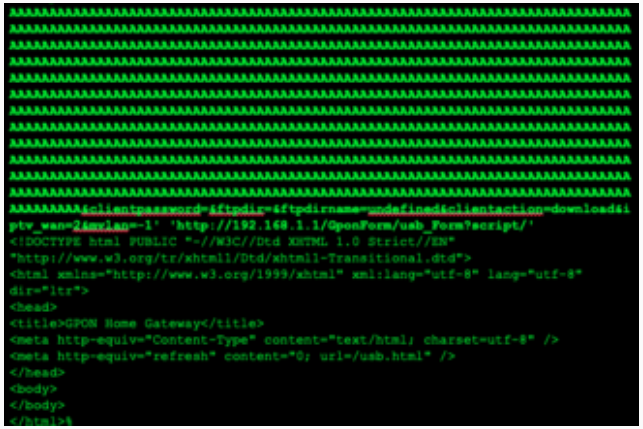
Follow

See responses (2)

More From Medium

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Related reads

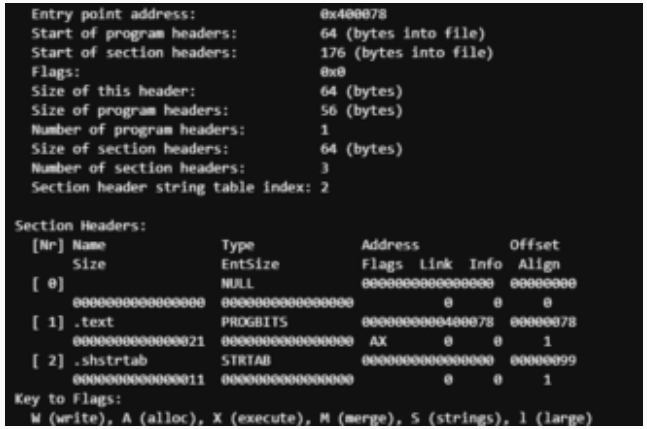


GPON Home Gateway RCE threatens tens of thousands users

Artem Metla in Tenabl...
Feb 27 · 6 min read

689

Related reads

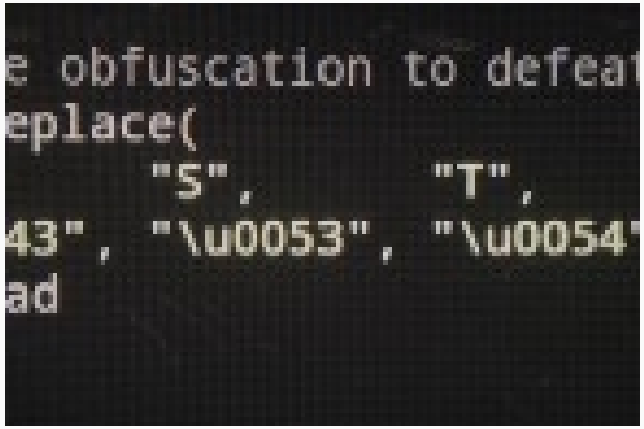


ELF Binary Mangling Part 1 — Concepts

【☆ゆう☆】
Aug 7, 2018 · 8 min read

184

Related reads



Making a Blind SQL Injection a Little Less Blind

TomNomNom
Jul 28, 2018 · 9 min read

1.1K

Create PDF in your applications with the Pdfcrowd HTML to PDF API
PDFCROWD