IEEE 754 and me – an experiment in *anonymizing* a PDF file

(I like to create words.)

In this page I explain a hack I did with the PDF file that contains the IEEE 754 standard.

I don't have access to the IEEE website, so I normally would have to pay to read this file. Except I know some people who have access. (I mean: access to some publications. Obviously if I enter the IEEE address in any browser I'll get some data back in my face.) So I just asked to them if they could get it for me.

So they did. And they gave it to me.

Except...

Except if I want to redistribute the file, I can't.

Well I can, obviously. It's just a file. I can put it on a website and there we are, it's shared.

But my problem is that the file contains some data from the people who gave it to me. So *they* would be in trouble if IEEE finds the file. They just look at it and "oh, but these are these people who distributed the file to the world. Let's sue them!"

I am close to these people. I just can't do that.

Except...

Except, you know, a file is a file. You can edit a file. You can modify it. You can extract stuff and put new stuff inside. You can do whatever you want with the file.

So you can remove the stuff that identify those who gave this particular PDF file to me.

That's what I did.

Here is some code to prove my claim.

Let me give some more details, will you my dear? It might be useful to young hackers out there. Some kind of lesson of hackery. I use the C language. I work under UNIX.

(Ah no. I don't distribute the clean IEEE 754 file.)
(I AM NOT A CRIMINAL!)
(Well...)

What is a PDF file?

A PDF file is a file that conforms to a given structure, namely the PDF one. 1

A description of it is freely available from Adobe. See here. (Adobe, like many big private corporations and many institutions, changes its internet website's structure from time to time. You might need to dig a bit to get the specifications. Alternatively you can look for "PDF specifications" with your favorite search engine.)

Since the PDF file I want to hack has a version of 1.4, I went there to download that. The current version, as of writing, is 1.7. Don't ask me why people like to change stuff when everything works fine. It's called "business" and it's something I'm completely unable to understand since it is so irrational.

So by reading this (big) documentation, you understand the structure of a PDF.

A PDF is made of several parts.

First part: the header. In our case, the first line of the file contains the version (*%PDF−1.4*) and the second contains a comment with byte greater than 127. These bytes are here, as prescribed by Adobe, to inform software out there that the file contains binary data, ie. bytes greater than 127. Why does it matter? Because the internet is an old beast and in the past (and maybe still now) some links or protocols or both send only bytes smaller than 127 in some cases. Putting high bytes early in the file forces them to switch to a full 8 bits mode, thus accepting bytes from 0 to 255 (this might not be correct, my memory is a bit, you know, well, bad).

Anyway, what matters to us here is that these bytes are normally chosen at random. But how do we know? So the file we are going to produce won't contain these numbers, but some others, just in case IEEE attach some meaning to them.

After the header you (generally) have objects, a lot of them. In our case we have more than 700 objects. What does an object look like? Well, something like the following.

```
 2 0 obj<</Type/Pages/Kids[3 0 R 4 0 R 5 0 R 6 0 R 7 0 R
endobj
```

You find a first number, which is the object's ID, starting from 1 and couting up, then a 0 (generally, can be used for stuff we don't care here) (all the objects in the IEEE file have a 0 here). Then you have *obj* then some stuff. Then, on the same or another line, you have *endobj* to, guess what? end the definition of an object.

Some objects may be compressed, which complicates slightly the extraction, but that's not very hard either. 2

Note that an objet's definition does not necessarily has to fit in one line. In our case IEEE put almost everything on the first line, which eases our life, but it's not mandatory.

After all the objects, you find the trailer, which contains something called *xref*. This stuff lists all the objects with their starting offset in the file. Then you find some more stuff useful for programs so you must keep them. One of the things in there is called */ID* and this one looks strange so we will remove it. Maybe IEEE tracks the file with the help of this string.

And that's it for a PDF file. It's nothing particulary tricky to understand and manipulate. You must record where in the file each object resides. And if you want to modify an object it changes the position of all the following objects. And this is a very bad design if I dare. But my bet is Adobe doesn't care I don't like their design.

Extracting objects

So we want to modify this IEEE file, to remove stuff put in there to identify who downloaded it, to "anonymize" the file.

How do we do that?

Well, after some thinking the conclusion pops up, obvious. We have to extract all the objects, modify those we have to and glue back everything together when we are done.

We can't modify the file directly.

Why not? I mean, we just could overwrite the data we don't want in and that's it, we are done.

But it's not that simple.

First, the size of a file is an information per se. So keeping it might be bad. We will *remove* objects (or more specifically objects' content since removing an object is a pain; you must then remove all references to it and this requires a lot of parsing).

Then some objects are compressed. Uncompressing them and storing them back changes their size.

(Well, after a bit of thinking, this does not appear to be that important. Emptying objects reduces their sizes. And an uncompressed object is described with less characters. So we could simply put an empty version of the object and the overwrite the remaining space with spaces.)

So we are up to extract object. And this is not very easy at first.

And here comes a bit of hacking. 3

The IEEE file is very friendly. Objects are all stored the same way. At the start of a line, you have the object's ID, a 0 and *obj.* And they start right after the header. So, just bypass the header, and read objects, up to a point where what you read is not an object anymore.

And since objects start with a simple line, just read one line. Check for the presence of *obj* in it. If it is present you have an object and keep going. If not you are done with the objects and you are up with the trailer.

And you know what? It's even simpler than that. After you smoke the header, read just one byte. If it's *x* you are in the trailer, if not it's an object.

How do I know that? By looking inside the file. A *cat file | less* is very useful. If you need to look at the binary data, I suggest using *od* like *od −tx1 −ta file | less* (do a *man od* to get more options).

So I looked at the file and just saw that to discriminate between the start of an object and the trailer I just needed to look for an *x*.

I normally would have to scroll a lot to get to the trailer when I examine the file. Fortunately, there is *tail* a program that only shows the last lines of a file. To see the last 400 lines of a file you do

tail –n 400 file | less. This saves a lot of scrolling time.

Here comes some code I wrote based on all that.

First, the main function.

```
 int main(void) { header();
while (obj()) dump_obj();
trailer();
return 0;
}
```

See, it's very short. It first deals with the header, then gets the objects, then the trailer.

By looking at the file, I know the header is only made of two lines, so here is *header.*

```
 void header(void) {


4

 FILE *f = fopen("header.obj", "wb");
ERR(f, "header");
read_line();
dump_buffer(f);
read_line();
dump_buffer(f);
if (fclose(f)) ERR(0, "header");
}
```

Very clear too.

For *obj* things are a bit more complicated.

```
 int obj(void) { int c;
c = getchar();
eof(c);
if (c == 'x') { ungetc(c, stdin);
return 0;
} no++;
size = 0;
put_char(c);
while (1) { c = getchar();
eof(c);
put_char(c);
if (c == '\n') if (endobj()) break;
} return 1;
}
```

As you can see, we first check for an *x*. Then we read everything until we see *endobj*. Once again, by inspecting the file, I saw that *endobj* only appears at the beginning of a line, so I look for it only when I received an "end of line" character, \ *n*.

What does *endobj* do? It just reads seven characters. (Seven and not six because I also saw that after *endobj* there is an "end of line" character which I include, just to get the correct start for the next object, or the *x* starting the trailer). If at some point it doesn't get the expected byte, it returns *0*, meaning we don't read *endobj* yet. Otherwise it returns *1*.

```
 int endobj(void) { int c;
c = getchar();
eof(c);
put_char(c);
if (c!='e') return 0;
c = getchar();
eof(c);
```

```
put_char(c);
if (c!='n') return 0;


5

 c = getchar();
c = getchar();
c = getchar();
c = getchar();
c = getchar();
return 1;
}
```

eof(c); eof(c); eof(c); eof(c); eof(c); put_char(c); put_char(c); put_char(c); put_char(c); put_char(c); if if if if if (c!='d') return 0; (c!='o') return 0; (c!='b') return 0; (c!='j') return 0; (c!='\n') return 0;

One important point here: if *endobj* appears in an object we are doomed. I played the game, because it makes my life much easier. And I won! No *endobj* in weird places in this file. Otherwise more complicated parsing would have been required. And you may not know it since you don't know me but I'm a very lazy person.

So that's it for the objects.

I pass the trailer's extraction. We simply read everything past the last object and put it in a file.

Uncompressing compressed objects

Once we extracted the objects we have to inspect their content, one by one, to decide what to do (keep the object, empty it or modify it).

And there comes a problem. Some objects are compressed. We see something like the following when using *less.*

```
331 0 obj<</Filter/FlateDecode/Length 10>>stream x<9C>+
```

Obviously, we have some more work to do.

And there comes something very important for a hacker. You just cannot face a problem like this one and solve it out of nothing, as by magic. You need something. You need enough knowledge to get the right intuition at how to attack the problem.

You need, in a word, *culture.*

So here you go read the PDF specifications to learn about this */Filter/FlateDecode* stuff. You read, page 46, that this encoding method uses *zlib/deflate.*

And there comes your culture. Because, what to do of this knowledge? What if you don't know what this *zlib* stuff is?

By reading others' code, by browsing a lot of software on your system and in the internet, by compiling by hand, by carefully looking at what happens then, and also, of course, by using software libraries, 6 you build a knowledge base in yourself. You become sort of an expert.

And there, in the case of *xlib/deflate*, I know what it is, and I know how to handle that. It means that the data has been compressed and that I can uncompress it using the popular *xlib* library, which everyone uses or almost. And it's almost certainly installed on your computer. Dozens and dozens of software you use on a daily basis use it. And you can use it too.

So you read */usr/include/zlib.h* (which is a bit of a pain, but you can manage it) and after a few trials and errors, you quickly write some code to uncompress a zlib's stream.

And you end up with something like the following, extracted from *deflate.c.* (Yes, I reversed inflate/deflate in my brain. Normally *deflate* is to decrease the size of a file, but I wrongly thought it

meant *decompress*. Poor french native speaker of me...)

```
 strm.next_in = (unsigned char *)buf;
strm.avail_in = size;
strm.next_out = (unsigned char *)out;
strm.avail_out = 4000000;
strm.zalloc = 0;
strm.zfree = 0;
if (inflateInit(&strm) != Z_OK) { fprintf(stderr, "infla
return 7;
} if (inflate(&strm, Z_FINISH) != Z_STREAM_END) { fprint
return 8;
} if (inflateEnd(&strm) != Z_OK) { fprintf(stderr, "infl
return 9;
}
```

It looks like magic? It's not. We setup a *strm* structure, then we call *inflateInit*, *inflate* and *inflateEnd* in order and check that the return value is correct.

I had troubles at first for *inflate*, not checking against *Z_STREAM_END* but *Z_OK*. I read */usr/include/zlib.h* too fast.

It might happen to you as well. You may read documention too fast, or understand it wrongly, leading to some time trying to understand what's wrong. It may become very furstrating after some time, especially when you don't hack just for yourself, with no time pressure, but when you work in a company, with a tight schedule. Key here is to be patient. In most of the cases, when using some new library, the problem comes from you. In some cases (I had the problem once a few years ago with *ptrace*; the manpage was wrong) the error is in the documentation. In some other cases it's a bug in the software. How to know 7 where the problem comes from so that you spend more time solving it and less time tracking it? It's hard to answer. There again culture plays a role. When you start your hackery's journey, you know nothing. So you spend a lot of time doing useless things. After a while

(several years) you start to understand what's going on and you focus on the important stuff.

In my case, I more or less applied the following reasoning to dig the problem. Since *zlib* is a well know library, since PDF files are also well established, it is unlikely that one or both of these beasts have a problem. The problem very certainly comes from me. What did I do wrong then? I read */usr/include/zlib.h*, maybe a bit too quickly. So maybe I should read it again, more carefully. So I did. And I then found that using *Z_OK* was inappropriate in my case. I had to use *Z_STREAM_END*.

That's how you look for a bug in your software. First, try to remember how you wrote it. Then value each step with a "confidence" number. You obtain a list of actions, the first being the most probably wrong and the last being practically 100% correct. (Normally you only have very few "most probably wrong" actions at a given point of your coding. Organizing your work is part of the journey too. It's a skill to get.) Then you review actions in order, looking carefully where you originally just looked quicky at some stuff.

You need to be humble. Most of the time *you* are the cause of the error. *You* didn't understand something clearly enough. *You* typed something wrong on your keyboard. *You* called some functions in the wrong way, in the wrong order, with wrong arguments. *You* didn't check return values correctly and implemented an appropriate response (like ending the program or displaying a meaningful message on the terminal).

But you must also trust your judgment. Sometimes you review all your possible mistakes and nothing comes out of it. So the problem is from somewhere else. It might be a library, the compiler, the language you use, the operting system, the hardware, anything. How do you know that you are not guilty and that the problem is from another part of the system? And which part? How do you decide you digged enough in one direction that you are almost confident the problem is not here?

There is no clear answer. One important thing to do when facing a problem is trying to decrease its size, extract it. You have a bug in your 1,000 lines program? Try to write a very small one that also exhibits the problem. The key point is the ability to extract a "model" on which you can think with full confidence. Something like "okay, I extracted a small version of my program; I am now 100% sure that if 8 I don't find a problem with this little program then my bigger program is not faulty but the sytem, somewhere, is. Then I'll try to narrow it to the correct part of the system, eliminating, one by one, every possible cause, up to the point I am left alone with the guilty beast. I will then shoot it in the head and I will be left with a perfectly correct program, solving my problem as it is intended."

And that takes a lot of time, a lot of trials and errors, a good analytical brain and a huge dose of patience and humility.

And here comes the *main* function of *deflate.c.*

```
 int main(void) { char *t;
char *l1, *l2;
int l;
int c;
z_stream strm;
read_line();
put_char(0);
if (!strstr(buf, "obj<<")) { fprintf(stderr, "WARNING: n
return 1;
} if (!strstr(buf, ">>stream")) { fprintf(stderr, "no st
return 2;
} if (!strstr(buf, "/Filter/FlateDecode")) { fprintf(std
return 3;
} if (!(t=strstr(buf, "/Length "))) { fprintf(stderr, "n
return 4;
} if (sscanf(t, "/Length %d", &l) != 1 || l <
0) { fprintf(stderr, "bad length found.\n");
```

```
return 5;
} fprintf(stderr, "INFO length %d\n", l);
l1 = strdup(buf);
ERR(l1,"strdup");
l2 = malloc(size);
ERR(l2,"malloc");
size = 0;
for (;
l;
l--) { c = getchar();
eof(c);
put_char(c);
} strm.next_in = (unsigned char *)buf;
strm.avail_in = size;
strm.next_out = (unsigned char *)out;
strm.avail_out = 4000000;
```



```
 strm.zalloc = 0;
strm.zfree = 0;
if (inflateInit(&strm) != Z_OK) { fprintf(stderr, "infla
return 7;
} if (inflate(&strm, Z_FINISH) != Z_STREAM_END) { fprint
return 8;
} if (inflateEnd(&strm) != Z_OK) { fprintf(stderr, "infl
return 9;
} fprintf(stderr, "INFO output bytes %ld\n", strm.total_
new_first_line(l1, l2, strm.total_out);
printf("%s", l2);
fwrite(out, strm.total_out, 1, stdout);
while (1) { c = getchar();
if (c == EOF) break;
printf("%c", c);
} fflush(stdout);
return 0;
}
```

We look at the first line in the file. By inspecting the extracted objects with *less* we found out that the first line contains all what is necessary to decide if the object is compressed or not. That's what we do, that's what all the tests do.

After that, if we indeed have a compressed file, we get the length of the compressed data, read this data, decompress it, and put everything back in a new, uncompressed, object. I don't show you *new_first_line*, it just removes */Filter/FlateDecode* and changes the */Length* stuff.

One interesting point here. The program will produce an empty output (and some messages on *stderr*). And here is how I run it on all the produced objects.

```
 for i in ????.obj;
do ./deflate <
$i >
out/$i;
done
```

So in *out/* some files will be empty. We can use that information to overwrite empty files (which thus are not compressed) with the current version. Here is how we do that. 10

```
 cd out;
find -empty -exec cp ../{} . \;
```

This line (and maybe the previous one) may look a bit magic.

There again we come back to *culture.* These commands (*for*, *find*, ...) and their arguments are "common knowledge" you acquire over time, by reading documentation, trying the command by yourself, and looking at how other people use them. There is no magic, just

culture.

Once it's done you are with objects in *out/*, that you finally can fully inspect and modify.

Modifying objects

In writing this webpage, I chose a linear order to present steps involved in the analysis and modification of a PDF file. The reality is that after I extracted the objects, and even before uncompressing them, I wrote *glue.c* that takes them all back and glues them together in a new PDF file. So maybe I should have put the "gluing objects" before this section. I don't know. When you hack you often write stuff "out of order" so as to have a few problems to solve at once. By writing *glue.c* early, just after the extraction, I could check that the extraction was correct, that the resulting PDF file obtained by collecting all the objects was similar to the original PDF. In fact, I needed to know if the extraction was correct. So obviously, the way to test that is to put them all back together and see what happens. And since I would need such a tool later anyway, why not write it right now? So that's why I wrote *glue.c* right after *extract.c*.

But let's look a bit at what needs to be done with our uncompressed objects.

I read them all. All the objects. More than seven hundreds of them. Just to see what was inside, what was needed to be removed to anonymize it.

I quickly found that some objects contained nothing but a text identifying who download the file.

A little surprise was to find several of those objects, all identical. Why not put only one? It's a clear loss of space. Maybe they thought it would be harder to remove. Maybe they just used some software that was suboptimal. It's not a big issue, but it forces us to do more than just editing by hand one file. We 11 have to come up with an

automatic solution.

So here is the magic line I quickly found.

```
 cd out;
for i in `grep -i limited *|cut -d : -f 1`;
do ../empty $i > $i;
done
```

And here comes *empty.c.*

```
 #include <stdio.h>
int main(int n, char **v) { int i;
sscanf(v[1], "%d", &i);
printf("%d 0 obj <</Length 0>>stream\nendstream\nendobj\
return 0;
}
```

The weird command line does a *grep* (very common and useful command in the UNIX world) over all the files, looking for "limited" (I first checked that "limited" only appeared in interesting files, and it did, so it's safe to discrimate files based on this string) and getting the file's name out of *grep*'s answer.

grep normally replies something like the following.

```
 0025.obj: A B C D limited E F G H
```

We have the file's name, double dot, and the line of file that matches what we look for.

We are interested in the file's name, and only that.

The wonderful *cut* command comes at help here. We ask it to cut

the first field *–f 1* by using the double dot separator *–d :*. The | (pipe) is a common mechanism under UNIX to plug one command to another. The output of the first command becomes the input of the other. You can chain many commands that way, manipulating data the way you want.

So after *grep* and *cut* we have a list of files. For each of those we call empty, that creates an empty object. It's important to pass the file's name to *empty* for the objet's number is written in the file, and that changes, obviously, with each object. 12

Looking at *empty.c* we see *sscanf(v[1], "%d", &i);* that will get the object's number from the file's name.

A last note on the command line. We see *for i in 'XX'* ..., with backquotes. Those backquotes are very useful. They transform the output of the command *XX* (or list of commands piped together) into arguments in the command line for the *for i* stuff.

It would have been possible to use *xargs* here, like in the following.

```
grep -i limited *|cut -d : -f 1|xargs ../empty
```

But I am more familiar with the first version (which may be slower, or not even work at all if too much data comes out of *XX* since command line's arguments have a limited size normally).

So, you have many ways to solve a problem. You don't need to learn all of them. What matters is to know one that works, and one that works fast enough. I could have done everything is C programs, which is my favorite horse. But I know some shell commands, and it's faster to use them than to write a C program from scratch. Maybe someone with a deeper knowledge of the shell and available commands would have done what I did faster. Maybe using other languages, like perl or python, would also speed up the coding of all this.

After some time you stick to your tools. Here, you must be careful. Are your tools good enough for your needs? Shouldn't you learn a few other tools? Where is the limit? When stop learning?

No clear answers there either. It's a matter of taste. And time. If you can solve a problem in one day with your tools and if you want to solve it in less than a week then it's okay. If you need one week of work for something you want to solve in one day you are in trouble. If you have no time constraints, do what you want. Hacking normally involves freedom, no tight schedule, so time should not be a problem. You normally care about beauty, elegance, this kind of stuff. Efficiency is a very subjective matter in the hacking world.

Removing images and colormaps

I don't trust images. Neither should you. It's easy to embed hidden data in images.

So I decided to remove them from the file. 13

Objects containing images are easy to find. There is */Image* on the first line (in the IEEE PDF file; remember that this file is very friendly for when you want to parse it; a lot of information is present on the first line).

I found two images. How did I process them?

I just changed */Width* and */Height*, set them to *1* and put a *0* in the stream.

There also was a colormap. I don't trust that either. Maybe (but I seriously doubt it) there is some hidden data there too. So I changed it to be full white, just in case.

Then I found some *ID* stuff here and there, which I also removed, and a file containing XML data, which I completely removed.

All in all, I tried to remove everything that was not necessary and which I thought would pose an anonymity problem.

Well, the process was not fully done.

I didn't change text's objects. I should have changed the spacing of commands in there. And maybe also their order. I should have changed the number of objects, their order in the file.

I mean, information is everywhere. You can hide a message in many many ways. If you want to do things properly, you have to change everything.

The best in our case would be to transform the PDF file by printing it as images, like *png* images for example, glue them in a file and distribute that file. Or extract text from them through OCR (optical character recognition), or even text directly from the PDF file, and generate a new PDF based on that.

Yes, to be really anonymous, you can't trust the file you download. You have to extract the data you care about, and be prepared that any other data may contain some information useful to identify you.

In my case, I don't care that much. I seriously doubt they go that far at IEEE to tag a poor file that costs a few dollars and which content is already known by many people, many websites, many sources.

But for images, no. I don't trust images. Period.

But this process could not be automatized. I would have had to do it each time I modify something in my programs. 14 So I just included the few objects in the sources, and in *transform.sh* I do the following.

```
 for i in *.n;
do cp $i out/`echo $i|sed -e "s/\.n//"`;
```

```
done
```

Once again, a bit of shell's magic.

This time, what we have to do is to overwrite files in *out/* with hand modified versions. These modified files are named with a *.n* suffix and are in the current directory.

Here the wonderful *sed* command shows up. I love this one. Even if I don't master it at all. So here I want to take a file's name like *0123.obj.n* and make it *out/0123.obj* so that *cp* can do its job (to copy one file to another). So I ask *sed* to look for *.n* and replace it with nothing. I use the backquote trick to get the name back on the command line and *cp* proceeds happily.

Note that I also could have put the modified files in a *mods* directory and do a simple *cp mods/* out/* but my brain, at that particular time, was not in this mood. And the command line above came out quickly, so I wasted no time here. I do this kind of manipulations so often that it's hardwired up there in the brain.

That explains why you sometimes see very convoluted solutions to simple problems. People are used to think in a way, to write code with a given style they're used to, that it ends up as a mess only them can understand.

How to protect against that?

Easy. First, solve your problem. Second, look at your solution with a critical eye. Third, rewrite your solution to be more elegant.

Obviously this all requires time and commitment. In a private company you generally don't do that. You have no time, schedule is tight, your thinking and coding costs a lot. So once you have a solution you stick to it. This is a very bad way to act. But hey, some people made some choices at points in history. Private corporations are one of those choices. They are clearly suboptimal ways to organize work. But they work, somehow, so as long as it

costs less to organize work this way and getting suboptimal solutions out of this setup, well, it will keep going. It's very hard to change such a big system.

This is some kind of hacking in itself, one might say. Change the world. On a large scale. 15 But we get out of programs and computers there, so let's stay with our simple PDF problem for the purpose of this webpage.

Recompressing objects

We now have our objects, cleaned up. No identification is possible anymore. (Unless IEEE uses complex schemes to mark its files, which I doubt a lot. They might think a PDF file is so complex no one would ever do the kind of things explained in this webpage.)

It's time to recreate a PDF file from those object.

A first attempt was made, where we simply put them back at this point, without compression whatsoever. The produced file was too big (around 2MB). So a recompression's step was necessary.

How to recompress? Well, straightforward. Just check if the object contains a stream. If yes, then compress the stream. If it turns out that the compressed version is smaller, put it in place.

Note that in the original file streams are compressed even if the compressed size is bigger. It happened for very small streams, but it happens nevertheless. What software they use I don't know, but it does not work very well, if I may.

With no more useless words, here comes the file *inflate.c* that compresses a file if it is compressible.

```
 #include #include #include #include #include #include

<stdio.h>
```

```
<stdlib.h>
<zlib.h>
<errno.h>
<string.h>
<ctype.h>

#define ERR do { printf("%s:%d: err\n", __FUNCTION__, __
exit(1);
} while (0) char *in;
int size;
int maxsize;
void put_char(char c) { if (size == maxsize) { in = real
if (!in) ERR;
} in[size] = c;
size++;
} char out[4000000];
```

16

```
 int main(int n, char **v) { int c;
FILE *f;
int l;
char *s;
char *t;
z_stream strm;
int i;
if (n != 2) { printf("gimme a file to process\n");
return 1;
} f = fopen(v[1], "r");
if (!f) ERR;
while (1) { c = getc(f);
if (c == EOF) { if (errno) ERR;
break;
} put_char(c);
} put_char(0);
fclose(f);
if (!(t=strstr(in, ">>stream"))){printf("%s has no strea
```

```
if (!s) ERR;
if (sscanf(s, "/Length %d", &l) != 1) ERR;
if (l == 0) {printf("%s has Length = 0, nothing done\n",
if (size >
4000000) ERR;
strm.next_in = (unsigned char *)t;
strm.avail_in = l;
strm.next_out = (unsigned char *)out;
strm.avail_out = 4000000;
strm.zalloc = 0;
strm.zfree = 0;
if (deflateInit(&strm, 9) != Z_OK) { fprintf(stderr, "de
return 7;
} if (deflate(&strm, Z_FINISH) != Z_STREAM_END) { fprint
return 8;
} if (deflateEnd(&strm) != Z_OK) { fprintf(stderr, "defl
return 9;
} if (strm.total_out >
l) { printf("compressed size (%ld) is more than uncompre
return 0;
} f = fopen(v[1], "w");
if (!f) ERR;
t = in;
while (t != s) { fprintf(f, "%c", *t);
t++;
}
```



```
 fprintf(f, "/Filter/FlateDecode/Length %ld", strm.total
while (!isdigit(*t)) t++;
while (isdigit(*t)) t++;
while (*t != '\n') { fprintf(f, "%c", *t);
t++;
} fprintf(f, "\n");
for (i=0;
i<strm.total_out;
```

```
i++) fprintf(f, "%c", out[i]);
fprintf(f, "\nendstream\nendobj\n");
fclose(f);
return 0;
}
```

Note that this time, a bit different from how *deflate.c* works, we output in the file itself, not on *stdout*, which eases the life in the shell's world. The simple following command does the trick very well.

```
 cd out;
for i in ????.obj;
do ../inflate $i;
done
```

I know, I know. My programs *deflate.c* and *inflate.c* have wrong names. But now it's done and an important notion in the hacking world is: laziness.

Gluing objects

And now for the last action: glue everything back together.

Things are a bit more complex. Or more specifically, verbose. It's not very complex.

Here is the *main* function.

```
 int main(void) { get_root_info(&f);
get_files(&f);
dump(&f);
return 0;
}
```

We need to get the "root" and "info" stuff from the trailer, to let the new PDF file start at the same position than the original one. What

does it mean for a PDF file to "start?" Well, we have a list of object. And one of them is the "root" of everything. This has to be specified in the trailer. For "info" I don't really know if it's necessary or not. I didn't want to check, so I just include it. 18

The function *get_root_info* is not very interesting. I won't show it here.

The function *get_files* is funny. It looks for all the *.obj* files in the current directory and puts them in a list. Then it sorts the list alphabetically, to get the object in the correct order for later on. Here it comes.

```
 void get_files(pdf_file *f) { DIR *d = opendir(".");
struct dirent *e;
struct stat s;
obj *o;
if (!d) ERR;
while (1) { e = readdir(d);
if (!e) { if (errno) ERR;
break;
} if (e->d_name[0] <
'0' || e->d_name[0] >
'9' || !strstr(e>d_name, ".obj")) continue;
o = new_obj(f);
o->file[FILENAME_MAX-1] = 0;
strncpy(o->file, e->d_name, FILENAME_MAX-1);
if (stat(e->d_name, &s)) ERR;
o->size = s.st_size;
} closedir(d);
qsort(f->o.o, f->o.size, sizeof(obj), cmp);
}
```

See the use of *qsort* to, well, sort the list of object. The *libc* is friendly, sometimes.

And we finish that with *dump* which processes objects in order. It

stores the number of bytes written so far in the *s* variable, and updates *o−>offset* for each object so that the trailer gets a correct value for the starting offset of each object.

```
 void dump(pdf_file *f) { FILE *fi;
size_t s = 0;
obj_list *l = &f->o;
obj *o;
int i;
int c;
s = printf("%%PDF-1.4\n%%\x88\x89\x90\x91\n");
for (i = 0;
i <
l->size;
i++) { o = &l->o[i];
o->offset = s;
fi = fopen(o->file, "r");
if (!fi) ERR;
```



```
 while (1) { c = getc(fi);
if (c == EOF) { if (errno) ERR;
break;
} if (fwrite(&c, 1, 1, stdout) != 1) ERR;
s++;
} if (fclose(fi)) ERR;
} printf("xref\n0 %d\n0000000000 65535 f \n", l->size +
for (i = 0;
i <
l->size;
i++) printf("%10.10d 00000 n \n", l>o[i].offset);
printf("trailer\n");
printf("<</Size %d/Root %d 0 R/Info %d 0 R>>\nstartxref\
fflush(stdout);
```

```
}
```

So this program is the most complex one, indeed. We need to store a list of objects. You just can't process data on the fly. The trailer forces us to "remember" some stuff for later use.

See the structures I had to create.

```
 typedef struct { char file[FILENAME_MAX];
size_t size;
size_t offset;
} obj;
typedef struct { obj *o;
int size;
int maxsize;
} obj_list;
typedef struct { int root;
int info;
obj_list o;
} pdf_file;
```

I wonder if a program using recursion rather than a list would not be possible. Would it be smaller? more efficient? more elegant? Well, this is not my style of programming, so I won't dig too much in that direction. People more in the functional programming may have some funny ideas there.

And how to pilot this program?

Easy.

20

```
 cd out;
../glue > TTT
```

And there you are. A nice, anonymous PDF file called *TTT*. Why this

strange name? Well, once again, laziness. I had to find one, I picked up something in my brain, for no particular reason. Habits, maybe. Weird habits? Maybe.

Automatizing the process

Obviously each time something new appears in there I had to run the process from scratch once again.

There were many bugs. Some at the syntaxic level, like calling a wrong function, wrong arguments, this kind of things. Some others were due to a bad understanding of the PDF specification. Some others to bad shell's knwoledge.

There are many bugs that pop up here and there. A hacker lives with those beasts, looks for them, eliminates them. You get used to it. Whatever procedure you put in yourself, you won't avoid bugs. So you need to develop here again a good culture of potential problems.

One nasty source of problems comes from the tools you use and the error messages they throw at your face. I specially think about the compiler. Sometimes you just forget a *;* and all in a sudden you get hundreds and hundreds of unrelated errors. Or you forget a *}*. Or some minor stuff like that.

There again you live with that. You compile often, so you are sure a compilation error can only originate from the few lines you introduced and not somewhere else.

And you debug harder bugs too. It may be frustrating sometimes. Getting good habits, a good culture, takes time and devotion. You have to bypass frustration and just keep going, up to success. I mean, this is not research. There is a solution, you know that from the beginning. So it's just a matter of doing things properly to get the expected result. That is what is so cool with hacking. The results are there, necessarily.

And so, to get back to automation, I could have written a "pilot" in C. But, traditionally, this is done in shell script.

We already saw the various shell's commands, but to recapitulate, here comes *transform.sh.*

```
 rm -f out/* cp trailer.obj out
```



```
 for i in ????.obj;
do ./deflate  $i > out/$i;
done (cd out;
find -empty -exec cp ../{} . \;) (cd out;
for i in `grep -i limited *|cut -d : -f 1`;
do ../empty $i > $i;
done) #(cd out;
for i in `grep -i limited *|cut -d : -f 1`;
do cp $i $i.emp;
done) for i in *.n;
do cp $i out/`echo $i|sed -e "s/\.n//"`;
done (cd out;
for i in ????.obj;
do ../inflate $i;
done) (cd out;
../glue > TTT)
```

Each line of this file would require several dozens of C code. That's why shell programming is something to learn too. It speeds things up for basic data manipulation.

But some treatments are beyond the shell and various command line's programs' ability and you need to write more complex program.

I do it in C, because that's what I know best. Several years of practice shaped my brain to easily think in C.

The main problem you face when you hack is bugs. And knowing a language, practicing it over the years, gives you familiarity. You know where to look for bugs. You know that when you looked here and found nothing then there is nothing wrong there. This knowledge you don't get for systems and languages you don't know. So you can't narrow the source of a problem. You can't become *certain.*

Well, that's it.

Conclusion

Do these programs "think?"

They manipulate data, extract information. As you and I do through our sensory system. Is it comparable?

Some might say they are not complex enough. They just apply a given procedure.

How are you certain you don't work this way, huh?

I don't pretend the programs here think. It's just not that simple to give a convincing *no.* Take that question as an opportunity to think about yourself, the world, life, the meaning (if any) in there and so on.

Or not. I mean, it could be seen as some kind of madness or sick attitude or whatever. It's just a tought experiment, based on "reality" (as long as a hack is "real").

Anyway, I'm done with this hack. Let's move on to something else. (The magic of hacking. It never ends.)

(By the way, what is IEEE 754? Short answer: dig the web. Longer answer: it's just a standard to represent floating point numbers in computers. And yes, normal people would think it's very boring. But I am not normal and 22 neither are you! Are you?)

Contact: sed@free.fr

Created: Mon, 06 Jul 2009 17:03:23 +0200
Last update: Mon, 06 Jul 2009 17:03:23 +0200