Eliot Jones
Thoughts of a developer

So you want to parse a PDF?

03 Aug, 2025

Suppose you have an appetite for tilting at windmills. Let's say you love pain. Well then why not write a PDF parser today?

The ideal world: how the specification should work

Conceptually parsing a PDF is fairly simple:

- First, locate the version header comment at the start of the file
- Next you need to locate the pointer to the cross-reference
- Then you can find all object offsets
- Finally you locate and build the trailer dictionary which points to the catalog dicitionary

Introduction to PDF objects

A PDF object wraps some valid PDF content, numbers, strings, dictionaries, etc., in an object and generation number. The content is surrounded by the `obj/endobj` markers, for example a simple number may have its own PDF object:

```
16 0 obj
620
endobj
```

This declares that object 16 with generation 0 contains the number 620.

A PDF file is effectively a graph of objects that may reference each other. Objects reference other objects by use of indirect references. These have the format "16 0 R" which indicates that the content should be found in object 16 (generation number 0). In this case that would point to the object 16 containing the number 620. It is up to producer applications to split file content into objects as they wish, though the specification requires that certain object types be indirect.

Finding the cross-reference offset

To avoid the need to scan the entire file, PDFs declare a cross-reference table (xref). This is an index pointing to where each object in the file lives.

Each file ends with a pointer to the cross-reference file:

```
<< %trailer >>
startxref
116
%%EOF
```

This tells the parser to jump to byte offset 116 to find the xref table (or stream). In theory this pointer is right at the end of the file, according to the specification:

> Applications should read a PDF file from its end. The last line of the file contains only the end-of-file marker, `%%EOF`

Though the specification says the `%%EOF` marker should be on the last line, in practice, things are much messier. For example, Adobe Acrobat only requires it to be within the last 1024 bytes. In real files it can appear anywhere.

In addition files encountered in the wild lacked a linebreak before the offset declaration, or had a typo, e.g. `startref`.

Let's assume you're able to find the declared cross-reference offset for now.

Finding all object offsets

At the specified offset you should find a well-formatted xref table:

```
xref
7 4
0000000000 65535 f
0000109882 00000 n
0000109933 00000 n
0000140066 00000 n
```

After the `xref` indicator appears, followed by a line break, the first object number and count of objects in the subsection are given. This means: start at object 7 and list 4 objects. Each line gives the byte offset, generation number, and status (n for in-use, f for free). From this, we know where to find objects 8-10 in the file.

So in the example above -- skipping the free entry for object 7 -- this xref table tells where to find the following objects:

- Object 8 (generation 0) at offset 109882
- Object 9 (generation 0) at offset 109933
- Object 10 (generation 0) at offset 140066

Note: files can have multiple xref tables or streams, linked by `/Prev` entries in their trailers.

Locating the trailer dictionary

Finally, above the startxref marker, you'll find the trailer dictionary. This provides key metadata, most importantly, where to find the root object. Once you have that, you can follow references and begin interpreting the content.

The real world: where your pain begins

Assuming everything is well behaved and you have a reasonable parser for PDF objects this is fairly simple. But you cannot assume everything is well behaved. That would be very foolish, foolish indeed. You're in PDF hell now. PDF isn't a specification, it's a social construct, it's a vibe. The more you struggle the deeper you sink. You live in the bog now, with the rest of us, far from the sight of God. If your parser expects files to obey the specification it will fail and people will think it is broken and pitiable. They will think that you are very silly.

The challenges of locating the xref pointer

We've already mentioned a few unexpected ways locating the pointer to the first cross-reference can fail:

- It is not at the end of the file, nor within the last 1024 bytes of the file.
- It is misspelled.
- It is not in the format you'd expect.

But assuming you find a pointer, that's where the real fun begins. Because the pointer is not your friend, it is the first lie, you don't appreciate how deep the rabbit hole goes. In screening 3977 files taken from the common crawl corpus at random we detected 23 files with a bad xref declaration. This works out to a roughly 0.5% failure rate in the sample set.

PDF content starting at a non-zero offset

In these files the `startxref` pointer is incorrect due to a non-zero PDF content start.

This happens when there's junk data before the `%PDF-` version header. This shifts every single byte offset in the file. For example, the declared startxref pointer might be 960, but the actual location is at 970 because of the 10 bytes of junk data at the beginning:

```
ten bytes!%PDF-1.4
%âãÏÓ
4 0 obj
(content follows)
endobj
% more content
```

```
xref
0 5
% ...
<< >>
startxref
960
%%EOF
```

In order to adjust for this you should capture the offset of the version header in your file. If the first pointer is incorrect you should also try the offset of the first pointer plus the content start offset. But you still need to check both.

This problem accounted for roughly 50% of errors in the sample set.

The pointer is in the middle of the xref table

For some files there is no content preceding the version header, however the pointer is still wrong and it points inside the xref table content at a random offset.

For example jumping directly to the specified offset takes you to this position:

```
endobj xref
0 246
0000000000 65535 f
0000184481 00000 n
00000<---
```

This was the case for roughly 5 files in the error set.

The pointer is 'close' to the xref

Similar to the previous error, here there is no version header offset but following the pointer takes you 'almost' to the xref. The most common cases were to be off by a single whitespace/newline, or in the `endobj` marker of the previous object:

```
endobj
--> xref
0 4
```

or:

```
-->endobj
xref
0 7
```

The pointer is correct but the xref offsets are incorrect

Sometimes the pointer correctly jumps to the `xref` marker but if you parse the object offsets from the table they are incorrect. The table offsets can also be incorrect when the xref offset is also incorrect.

It can also be the case that offsets are correct for some objects in the table but wrong for others. This was the case for file 0002544.pdf in the sample set which had an initial pointer off-by-7. The locations in the xref table's first subsection were correct, then offsets that were off-by-4 bytes for subsequent subsections.

The first pointer is correct but the previous offset is incorrect

When a file has been modified the file's trailer (or xref stream dictionary) can contain a `/Prev` pointer. This is used to construct a chain of xref tables and streams. Several files had correct initial pointers however when parsing the trailer's previous offset the second location was incorrect. One file contained a value of `0` for the previous pointer which indicates that it had incorrectly written the default value, rather than an offset.

The xref table is not well formatted

Beyond the xref pointer issues seen in the sample set, the table structure itself can be malformed in unexpected ways.

The following examples were reported as Github issues for PdfPig.

No linebreak after `xref`, for example:

```
xref5 2
0000000000 65535 f
0000134883 00000 n
```

More object entries in a subsection than declared in the header, for example if only 2 objects are declared the table can contain more:

```
xref
0 2
0000000000 65535 f
0000000230 00000 n
0000000520 00000 n
0000001000 00000 n
```

Garbage in the middle of the table, for example:

```
xref
0 2
0000000000 65535 f
0000455.8483a a010 00000 n
```

Conclusion

We looked at how parsing a PDF should proceed according to the specification. We then compared this with a survey of sample files where we saw a 0.5% error rate due to non-compliant files. All tested PDF viewers (PDF.js, Adobe, Sumatra) were able to open these files because most parsers are extended to support non-compliant files.

This serves as a brief survey of the challenges of parsing a single part of the PDF specification (22 pages out of 1,300 total from version 1.7).

Recent Posts

Other Sites