Frank Seifferth

Combining pdf and zip proof of concept

24th June 2025

As I will outline in an upcoming article in Paged Out! (forthcoming), it is actually possible to create a pdf document that contains a zip archive both as a pdf attachment (Adobe 2003, 156–159) and using Julia Wolf's famous polyglot technique for merging the two file formats (Wolf 2013; Albertini 2019); both at the same time and using a single representation of the embedded/attached data. All that is required to achieve this feat is to create the pdf attachment in such a way that detaching it does not change any relevant offsets within the zip archive. This blog post explains the technical details of how this can be done in practice and thus serves as a proof of concept that illustrates the feasibility of the approach. Not least by employing the same technique to assemble this blog post itself, which — rather conveniently — is also published as a pdf document.

The rest of this blog post is divided into five steps. In the first two steps, we simply create an entirely normal zip archive and pdf document. The third step is to pad the zip with null characters so that both the pdf and the zip have the exact same size. In step four we then create a zlib-compressed stream that, when decompressed, becomes the padded zip archive created in step three. The fifth step, finally, is to embed this zlib stream inside the pdf document in order to create a polyglot file that both *is* and *contains* the same zip.

tl;dr

```
$ unzip pdf-zip-poc.pdf
Archive:  pdf-zip-poc.pdf
  inflating: pdf-zip-poc.md
  inflating: references.bib
  inflating: blog-template.latex
  inflating: embed_zlib.py
  inflating: pad_zip.py
  inflating: pdf_offsets.py
  inflating: to_zlib.py
  inflating: Makefile
$ make
test -f boring.zip && rm -v boring.zip || true
zip boring.zip pdf-zip-poc.md references.bib [...]
  adding: pdf-zip-poc.md (deflated 65%)
  adding: references.bib (deflated 58%)
[...]
$ pdfdetach polyglot.pdf -save 1 -o detached.zip
$ diff <(zipinfo polyglot.pdf) <(zipinfo detached.zip)
1c1
< Archive:  polyglot.pdf
---
> Archive:  detached.zip
```

Step 1: zip

In order to combine pdf and zip in the way described above, the first thing we need to do is to create a pdf document and a zip archive. Let's start with the zip archive, using this blog post's own source code to illustrate:

```
$ zip boring.zip *.md *.bib *.latex *.py Makefile
  adding: pdf-zip-poc.md (deflated 65%)
  adding: references.bib (deflated 58%)
  adding: blog-template.latex (deflated 58%)
  adding: embed_zlib.py (deflated 57%)
  adding: pad_zip.py (deflated 52%)
  adding: pdf_offsets.py (deflated 34%)
  adding: to_zlib.py (deflated 56%)
  adding: Makefile (deflated 55%)
```

Since we still need to work on the zip archive before we can actually add it to the pdf, we also create a placeholder file that is slightly larger than ‘`boring.zip`’:

```
$ wc -c boring.zip
16507 boring.zip
$ head -c 18507 /dev/zero >source.zip
$ wc -c source.zip
18507 source.zip
```

Step 2: pdf

For starters, we use our placeholder ‘`source.zip`’ as a pdf attachment. While there are many ways to produce a pdf that includes an attachment, here we will use poppler’s ‘`pdfattach`’. Compared to other methods — such as using a latex package like ‘`attachfile`’, ‘`attachfile2`’ or ‘`embedfile`’ — ‘`pdfattach`’ is very simple. It merely adds an incremental update containing the attachment to an existing pdf; and it does not even compress the attachment in any way. For our purposes, this simplicity has two significant benefits: First, the incremental update approach ensures that the attached file is placed near the end of the pdf. This will become important when creating a polyglot since zip comments have a maximum size of 65535 bytes. And second, since ‘`pdfattach`’ adds the attachment as an uncompressed stream, it gives us fine-grained control over how many bytes are allocated for our attachment, which will become important once we replace the placeholder with the actual zip.

Since to run ‘`pdfattach`’ we need an existing pdf, we first invoke the following command to compile a regular pdf document without attachments:

```
$ mkpdf pdf-zip-poc.md -o boring.pdf
Converting files to latex
Postprocessing pandoc output
Running latexmk ...
Exporting output: boring.pdf
```

In case you don't want to compile the pdf from source, but you happen to have a copy of the polyglot file that is this blog post, an alternative to running the command specified above would be to strip the incremental update from the final polyglot like this:

```
$ sed '/^%%EOF/q' pdf-zip-poc.pdf >boring.pdf
```

In a second step, we then use '`pdfattach`' to add our placeholder '`source.zip`' as an uncompressed pdf attachment:

```
$ pdfattach boring.pdf source.zip placeholder.pdf
```

Finally, we need to determine the precise location of our placeholder within the pdf document. Since our placeholder is just a very long sequence of null characters, we can use a rather straightforward python script to display how many bytes are located before, within and after the placeholder:

```python
#!/usr/bin/env python3
#
# Usage: ./pdf_offsets.py FILE

import sys

with open(sys.argv[1], 'rb') as f:
    placeholder_pdf = f.read()
before = placeholder_pdf.find(50*b'\x00')
after = placeholder_pdf[::-1].find(50*b'\x00')
within = len(placeholder_pdf) - before - after
print(before, within, after)
```

When executed, this script produces the following numbers:

```
$ ./pdf_offsets.py placeholder.pdf
96995 18507 498
```

Step 3: pad zip with zeroes

Now that we know the offsets we are aiming for, we can start preparing the zip archive for inclusion in the polyglot. As a first step, we simply pad the beginning and end of the zip archive with null characters. This allows us to make the zip archive's size match that of the pdf document and to place the actual contents of the zip archive within the same byte range allocated for our placeholder attachment. Since padding the zip with null characters involves a few important details, we use the following python script to perform this operation:

```python
#!/usr/bin/env python3
#
# Usage: ./pad_zip.py INFILE OUTFILE BEFORE WITHIN AFTER

import sys, struct

with open(sys.argv[1], 'rb') as f:
    boring_zip = f.read()
before, within, after = map(int, sys.argv[3:6])
# In step 5 (embed_zlib.py) we will need some space to
# update the pdf object metadata. We gain this space
# by virtually shifting the start of the embedded zip
# archive 500 bytes to the right, thereby giving us
# 500 bytes of our initial placeholder for additional
# pdf metadata:
before += 500; within -= 500
# Furthermore, we want to use a zip comment to hide
# the pdf contents following the zip; so we need to
# know the length of this comment:
commentlength = within - len(boring_zip) + after
# There are also two failure modes we can guard
# against at this point. First, our initial zip
# needs to fit within the space allocated for the
# placeholder. Second, the zip comment's length is
# stored as an unsigned 16-bit integer, which results
# in a maximum length of such zip comments.
if not len(boring_zip) < within: raise Exception()
if not commentlength < 2**16: raise Exception()
```

```
# Additionally, we will assume that the input zip has
# no comment, which means that the last two bytes will
# be zero; even though the last two bytes being zero
# does by no means guarantee that there is no comment.
if boring_zip[-2:] != b'\x00\x00': raise Exception()

# Finally, we can create the padded zip archive. We
# simply use null characters for the padding we add both
# before and after the actual zip. The last two bytes
# of a zip archive trailer specify the length of its
# comment. In our case --- since our zip archive has
# no comment --- those are the last two bytes in the
# file, which are \x00\x00. We remove those two bytes
# and instead insert the comment length we calculated
# above, represented as a little-endian 16-bit unsigned
# integer which we create using 'struct.pack'.
with open(sys.argv[2], 'wb') as f:
    f.write(before * b'\x00')
    f.write(boring_zip[:-2])
    f.write(struct.pack('<H', commentlength))
    f.write(commentlength * b'\x00')
```

We invoke this script with the three lengths calculated at the end of step two in order to create a zip archive padded with null characters:

```
$ ./pad_zip.py boring.zip padded.broken.zip \
                                        96995 18507 498
$ wc -c placeholder.pdf; wc -c padded.broken.zip
116000 placeholder.pdf
116000 padded.broken.zip
```

While the resulting ‘`padded.broken.zip`’ contains the right padding — with the final padding even marked as a zip comment already — the zip archive is still invalid since we did not adjust the offsets pointing to the individual files within the zip. Luckily, we can fix these offsets rather easily by invoking ‘`zip -F`’:

```
$ zip -F padded.broken.zip --out padded.zip
Fix archive (-F) - assume mostly intact archive
Zip entry offsets appear off by 97495 bytes - correcting...
```

```
copying: pdf-zip-poc.md
copying: references.bib
copying: blog-template.latex
copying: embed_zlib.py
copying: pad_zip.py
copying: pdf_offsets.py
copying: to_zlib.py
copying: Makefile
```

This, finally, gives us a valid zip archive that has the same size as our '`placeholder.pdf`' and whose actual contents lie comfortably within the byte range we allocated for our pdf attachment.

Step 4: zlib stream

The next problem we need to address is how to embed this zip archive in our pdf such that (1) the the actual contents of the zip archive are stored in an uncompressed manner and (2) the detached zip archive has the same size as the pdf it is attached to. Luckily, we can use a neat combination of pdf and zlib semantics to achieve just that. On the one hand, pdf objects can be compressed with zlib (Adobe 2003, 46), allowing us to fulfil condition 2. On the other hand, zlib-compressed streams can themselves contain blocks of uncompressed data (Deutsch and Gailly 1996, 1; Deutsch 1996, 11). This gives us a neat way to simultaneously fulfil condition 1. All we need to do is to extract the actual contents from our '`padded.zip`' and then place them inside a zlib-compressed stream; marked as an uncompressed block and placed between two compressed blocks that can be decompressed to reproduce the padding with null characters which we used in creating '`padded.zip`'.

Since creating this special zlib-compressed stream involves meddling with some rather intricate details of the binary stream, we use yet another python script to perform this operation:

```python
#!/usr/bin/env python3
#
# Usage: ./to_zlib.py INFILE OUTFILE BEFORE WITHIN AFTER

import sys, struct, zlib
```

```python
with open(sys.argv[1], 'rb') as f:
    padded_zip = f.read()
before, within, after = map(int, sys.argv[3:6])
# Virtually shift start of embedded file right (same as
# pad_zip.py) and then extract the actual zip_content.
# Note that the zip_content extracted in this way
# includes some null characters at the end, which, for
# our purposes, is entirely fine. Including at least one
# null character is even advisable since this character
# also hides the zip comment from most applications.
before += 500; within -= 500
zip_content = padded_zip[before:before+within-500]
# Ensure there is enough space for our data structures
if padded_zip[before+within-500:][:500] != 500*b'\x00':
    raise Exception("WITHIN is too small")
# Ensure that zip_content fits in one uncompressed block
if not len(zip_content) < 2**16:
    raise Exception("zip_content is too long")

# Now we start assembling our zlib-compressed stream.
# We start by manually writing the magic number.
z = b'\x78\xda'       # zlib magic number
# Next, we create the left padding by compressing
# BEFORE null characters. Note that 'zlib.compress'
# creates a whole zlib stream, not just one block of
# data. Therefore, the output contains another magic
# number (the first two bytes) and a checksum (the last
# four bytes) which we remove from the output.
pad_left = zlib.compress(before*b'\x00', level=9)[2:-4]
# Also note that the first bit of a block indicates
# whether or not this is the last block in the
# stream. Since we want to manually add more blocks
# later on, we flip the bit indicating that this is
# the last block before adding it to our stream.
z += (pad_left[0] ^ 0b00000001).to_bytes(1)
z += pad_left[1:]
# Adding the uncompressed block next is a little
# tricky. According to zlib/deflate semantics, an
# uncompressed block starts with the three bits '100'
# if it is the last block or with the three bits '000'
# if it is not the last block. For uncompressed blocks
# --- as well as, apparently, for the last block in
# a stream --- the space up to the next byte boundary
# is then padded with zeroes. However, the compressed
# blocks are not byte-aligned, so we do not know if
# the compressed block we just added already contains
# the three bits '000' that would indicate the start
# of an uncompressed block or if it does not. In order
# to figure this out, we use the decompressor to check
# whether or not we still need to add those extra bits.
try:
    zlib.decompress(z+b'\x00\x00\x00\xff\xff')
except zlib.error as e:
    if str(e).startswith('Error -3'):
        # "invalid stored block lengths", indicating
        # that we do not need the extra byte
        pass
    elif str(e).startswith('Error -5'):
        # "incomplete or truncated stream", indicating
        # that the extra byte is necessary
        z += b'\x00'
    else:
        raise e
# Now that we have correctly started the block of
# uncompressed data, we can specify its length by
# filling the LEN and NLEN fields.
z += struct.pack('<H', len(zip_content))
z += struct.pack('<H', len(zip_content) ^ 0xffff)
# Afterwards, we can add the actual zip_content of the
# uncompressed block.
z += zip_content
# And finally, we can add the right padding (which,
# again, will be compressed) and the checksum. Since the
# right padding is indeed the last block in the stream,
# we do not even need to perform the bit flip used for
# the left padding.
z += zlib.compress((after+500)*b'\x00', level=9)[2:-4]
z += zlib.adler32(padded_zip).to_bytes(4)
with open(sys.argv[2], 'wb') as f:
    f.write(z)
```

We can invoke this script as follows to generate a zlib stream:

```
$ ./to_zlib.py padded.zip padded.zlib 96995 18507 498
```

And we can use '`zlib-flate`' and '`diff`' to confirm that our '`padded.zlib`' stream correctly decompresses to '`padded.zip`':

```
$ zlib-flate -uncompress <padded.zlib |
                    diff --report-identical - padded.zip
Files - and padded.zip are identical
```

Step 5: polyglot

Now that we have created a zlib-compressed stream that decompresses to '`padded.zip`', we can finally create our polyglot by replacing the placeholder we have put in '`placeholder.pdf`'. If we open '`placeholder.pdf`' in a text editor and search for a long sequence of null characters, we can see that the contents of our pdf attachment are stored in an object that looks roughly like this:

```
190 0 obj
<</Length 18507 /Params <</Size 18507 >> >> stream
[[ null characters ]]
endstream
endobj
```

What we see here is a pretty common syntactic construct in pdf. First of all, our stream of null characters is embedded within a pdf object that starts with '`190 0 obj`' and ends with '`endobj`'. Inside this pdf object, we first find the metadata dictionary; which is simply a list of key-value pairs placed between '`<<`' and '`>>`'. The contents of this metadata dictionary are pretty straightforward to understand. First we find the key '`/Length`' followed by the length of the binary stream. Next we find the key '`/Params`' whose value is another dictionary. This nested dictionary, in turn, contains another key-value pair that specifies the file size of the detached attachment. Since we stored our placeholder without any form of compression, the size of the detached file matches that of the embedded stream. Below we find the 18507 null characters between the ‘stream’ and ‘endstream’ keywords. Note that the ‘stream’ keyword must be followed by a single linefeed character that is not interpreted as forming part of the stream itself. I am not entirely sure if the ‘endstream’ keyword must be preceded by a linefeed character, but it certainly does not hurt to place one there as well.

In order to replace the placeholder with the zlib stream we created in step four, we need to make some adjustments to this pdf object. For one thing, we need to replace the null characters between the ‘stream’ and ‘endstream’ keywords with the contents of ‘padded.zlib’. Additionally, we need to update the object’s metadata dictionary. Since both our zlib stream and the uncompressed ‘padded.zip’ have a different size than the placeholder we initially used to create the pdf attachment, we need to adjust the values for both ‘/Length’ and ‘/Size’ to match the size of ‘padded.zlib’ and ‘padded.zip’ respectively. Additionally, we need to add the new key-value pair ‘/Filter/FlateDecode’ to the metadata dictionary to indicate that the following stream contains zlib-compressed data. Once we make these changes, our updated pdf object should look roughly like this:

```
190 0 obj
<< /Filter/FlateDecode /Length 17639
   /Params << /Size 116000 >>
>>
stream
[[ zlib stream ]]
endstream
endobj
```

In order to break neither pdf nor zip semantics by making those modifications, we also need to keep a number of additional constraints in mind. First, the offset of the zip header’s magic number ‘PK’ within the polyglot needs to match the offset of that header in ‘padded.zip’. Second, the pdf contains an ‘xref’ table specifying the absolute offset of each object within that pdf. Therefore, we should take care to preserve both the offset of the object we modify — i. e. the offset of ‘190 0 obj’ — and the offset of everything located after the ‘endobj’ keyword. Luckily, whitespace is insignificant in many places inside a pdf document, so we can safely place some additional spaces inside the metadata dictionary and after the ‘`endobj`’ keyword. With these constraints in mind, we can write a python script that replaces the placeholder attachment in ‘`placeholder.pdf`’ with the contents of ‘`padded.zlib`’:

```
#!/usr/bin/env python3
#
# Usage: ./embed_zlib.py INPDF ZLIBFILE OUTPDF

import sys, zlib

with open(sys.argv[1], 'rb') as f:
    placeholder_pdf = f.read()
with open(sys.argv[2], 'rb') as f:
    padded_zlib = f.read()
    padded_zip = zlib.decompress(padded_zlib)

# The first thing we need to figure out is the location
# (start and end) of our placeholder attachment. In
# contrast to what we calculated in pdf_offsets.py,
# however, this time we are looking for the whole
# pdf object and not only for the sequence of null
# characters. Therefore, finding the start and end
# of the placeholder is slightly more complicated
# this time.
#
# To find the start of our pdf object, we first look
# for the sequence of null characters:
objstart = placeholder_pdf.find(50*b'\x00')
# Then we walk left until we find the 'obj' keyword:
while placeholder_pdf[objstart-4:objstart] != b' obj':
    objstart -= 1
# To find the end of our pdf object, we simply look for
# the first 'endobj' keyword located after the start
# of the object; and then we add 6 (the length of the
# keyword itself):
objend = placeholder_pdf.find(b'endobj', objstart) + 6

# The last piece of information we need to figure
# out is where to place the start of the zlib stream
# such that the zip's magic number 'PK' is located at
# the same offset in both padded.zip and in our final
# polyglot. We can find this offset by subtracting the
# length of what precedes 'PK' in the zlib stream from
# where 'PK' is meant to be located in the uncompressed
# file. Finally, we subtract the length of the 'stream'
# keyword and the associated newline characters.
streamstart = padded_zip.find(b'PK') -\
              padded_zlib.find(b'PK') -\
              len(b'\nstream\n')

# Now that we know all the offsets, we can finally
# assemble the polyglot file. We start by adding
# everything up to the object we want to modify,
# followed by a newline character (which we skipped
# over while looking for the 'obj' keyword):
pdf = placeholder_pdf[:objstart] + b'\n'
# Next, we add our updated metadata dictionary:
pdf += ('<< /Filter/FlateDecode /Length {}\n'\
        '   /Params << /Size {} >>\n'\
        '>>\n')\
            .format(len(padded_zlib), len(padded_zip))\
            .encode('ascii')
# Afterwards, we correctly align the start of the zlib
# stream by adding as much whitespace as we need:
pdf += (streamstart - len(pdf)) * b'\x20'
pdf += b'\nstream\n'
pdf += padded_zlib
pdf += b'\nendstream\nendobj\n'
# And just to make sure that we actually got the
# alignment right, we double-check it like this:
if pdf.find(b'PK', objstart) != padded_zip.find(b'PK'):
    raise Exception('broken magic number alignment')
# As a last step, we add some more padding and then add
# everything that was originally located after the end
# of the object we just modified:
pdf += (objend - len(pdf) - 1) * b'\x20' + b'\n'
pdf += placeholder_pdf[objend:]
# Once again we can double-check our alignment:
if len(pdf) != len(placeholder_pdf):
    raise Exception('broken pdf trailer alignment')
```

```
# Et voilà: A pdf/zip polyglot where the zip is valid
# both before and after detaching the pdf attachment.
with open(sys.argv[3], 'wb') as f:
    f.write(pdf)
```

Invoking this script allows us to create the final pdf/zip polyglot:

```
$ ./embed_zlib.py \
                placeholder.pdf padded.zlib polyglot.pdf
```

We can double-check the validity of our result by running commands like ‘`zip -T`’ and ‘`qpdf --check`’:

```
$ zip -T polyglot.pdf
test of polyglot.pdf OK
$ qpdf --check polyglot.pdf | fold -s -w40
checking polyglot.pdf
PDF Version: 1.5
File is not encrypted
File is not linearized
No syntax or stream encoding errors
found; the file may still contain
errors that qpdf cannot detect
```

We can also use ‘`pdfdetach`’ to confirm that the pdf does indeed contain an attached file; and we can detach the zip archive to confirm that the detached zip is not only valid, but that it even contains the exact same contents the polyglot itself contains when interpreted as zip:

```
$ pdfdetach -list polyglot.pdf
1 embedded files
1: source.zip
$ pdfdetach polyglot.pdf -save 1 -o detached.zip
$ zip -T detached.zip
test of detached.zip OK
$ diff <(zipinfo polyglot.pdf) <(zipinfo detached.zip)
1c1
< Archive:  polyglot.pdf
---
> Archive:  detached.zip
```

References

Adobe Systems Incorporated. 2003. *PDF Reference fourth edition: Adobe Portable Document Format Version 1.5*. https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/pdfreference1.5_v6.pdf.

Albertini, Ange. 2019. 'Adding any external data to any PDF'. *Paged Out!* 1:17. https://pagedout.institute/download/PagedOut_001_beta1.pdf#page=16.

Deutsch, L. Peter. 1996. 'DEFLATE Compressed Data Format Specification version 1.3'. RFC 1951. https://www.rfc-editor.org/rfc/pdfrfc/rfc1951.txt.pdf.

Deutsch, L. Peter and Jean-Loup Gailly. 1996. 'ZLIB Compressed Data Format Specification version 3.3'. RFC 1950. https://www.rfc-editor.org/rfc/pdfrfc/rfc1950.txt.pdf.

Seifferth, Frank. Forthcoming. 'Re: Adding any external data to any PDF'. *Paged Out!* 7. https://tilde.club/~seifferth/redirect/pdf-zip-article/.

Wolf, Julia. 2013. 'This ZIP is also a PDF'. *PoC||GTFO* 0x01:11–12. https://www.alchemistowl.org/pocorgtfo/pocorgtfo01.pdf#page=11.