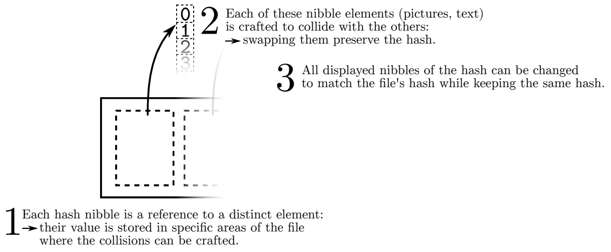
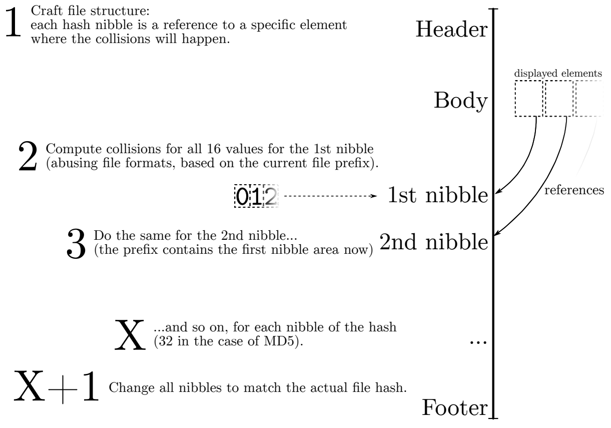
14:10 A PDF That Shows Its Own MD5

by Mako

Even though MD5 is quite broken, you might easily assume that creating a file that contains its own MD5 is impossible. After all, surely changing the file would change its MD5? Let's honor this publication's fine history of PDF tricks by creating a PDF file that displays its own MD5 hash when viewed.


2 Each of these nibble elements (pictures, text) is crafted to collide with the others: ➛swapping them preserve the hash.
3 All displayed nibbles of the hash can be changed to match the file's hash while keeping the same hash.
1 Each hash nibble is a reference to a distinct element: ➛their value is stored in specific areas of the file where the collisions can be crafted.

Our tactic will be to make each digit of the MD5 checksum a separate JPEG image, and make the MD5 hashes of all 16 possible images collide to the same value. We can then swap out images to display any combination of digits without affecting the file's MD5. This requires 15 collisions per digit, and since they depend on the MD5 of the preceding part of the document, we need to do this for each digit, for a total of $15 \times 32 = 480$ collisions. With a few compute-months of power we could just append chosen-prefix collisions to whatever images we liked and be done with it, but that's too slow. If we could make do with faster shared-prefix MD5 collisions — for example Marc Stevens' Fastcoll[33] — we could be finished in an hour.


1 Craft file structure: each hash nibble is a reference to a specific element where the collisions will happen.
Header
displayed elements
Body
2 Compute collisions for all 16 values for the 1st nibble (abusing file formats, based on the current file prefix).
012
1st nibble
references
3 Do the same for the 2nd nibble... (the prefix contains the first nibble area now)
2nd nibble
X ...and so on, for each nibble of the hash (32 in the case of MD5).
...
X+1 Change all nibbles to match the actual file hash.
Footer

This adds some restrictions. Everything other than the pairs of collision blocks must now be the same. Furthermore, the two versions of the first collision block have a fixed relationship, as shown in Figure 10.

If we could only get one of those bits to be in the length field of a JPEG comment marker, we could take loving inspiration from Ange Albertini's trick in the SHAttered attack, colorfully explained by Hector Martin[34] in Figure 11, to display two different images.

Unfortunately, they're in the middle of the collision block, and worse, those message words are being used to satisfy these constraints on Q[5], Q[12] and Q[15]:[35]

```
Q[5]  = 01000^01 11111111 11111111 11^^10^^
Q[12] = 0!0....0 ..!..01. ..1...1. 1.......
Q[15] = 1.0....0 .......! 1....... ....0...
```

. is don't-care,
ˆ is same as previous Q,
! is inverted from previous Q.

Hmmm. Q[15] is pretty lightly constrained. Maybe we could just set $m[14] = (m[14]\&$`0xff000000`$)|$`0x01feff` and see what it does to $Q[15]$. That'd give a JPEG comment of length 256-383 bytes on one side and 128 bytes longer on the other, and we can try just generating new sets of values until they meet the constraints. Luckily this works often enough to be practical, though there are probably more elegant approaches.

Now we can start colliding JPEGs! The structure is quite simple: we begin with an `FF D8` start-of-image marker and the parts that are identical in all our images, such as the JFIF APP0 segment, then add a JPEG comment that will end at exactly byte 56 of our collision block. After padding to a 64-byte block boundary and creating a collision, we finally have two partial files with identical MD5 values but different JPEG comment lengths.

From here it's straight sailing. In the short-comment version, the next JPEG marker parsed is a

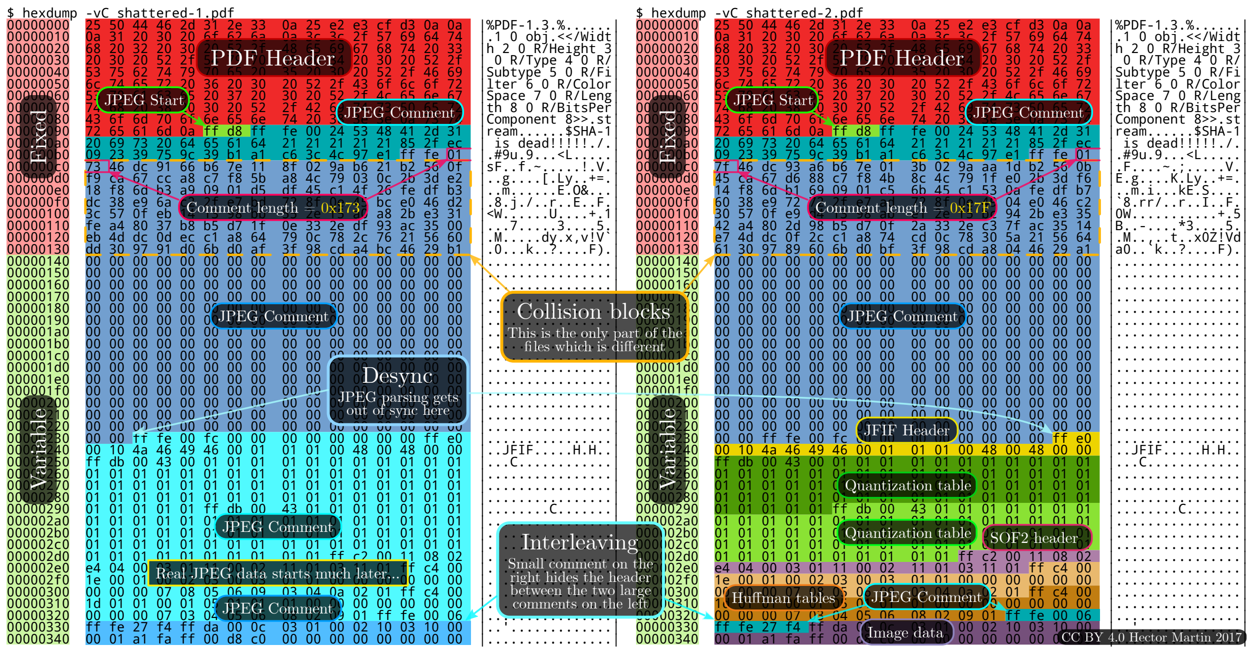
[33]`unzip pocorgtfo14.pdf fastcoll-v1.0.0.5-1.zip`

[34]See https://twitter.com/marcan42/status/835175023425966080

[35]*If these constraints look like voodoo or hoodoo to you, please* `unzip pocorgtfo14.pdf md5-1block-collision.pdf stevensthesis.pdf` *and read Marc Stevens' papers on how the collisions are formed. Don't expect to learn all of his magic in just a weekend.* —PML

$$
\begin{gathered}
block_b[4] = block_a[4] + (1 << 31);\\
block_b[11] = block_a[11] + (1 << 15);\\
block_b[14] = block_a[14] + (1 << 31);\\
\text{(rest of block is unchanged)}
\end{gathered}
$$

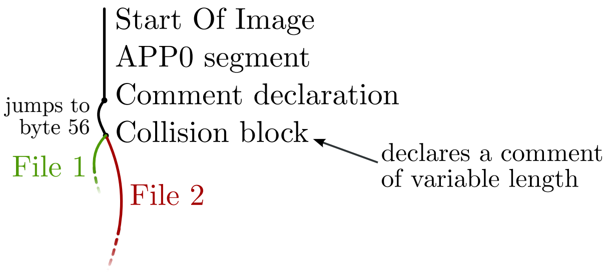
Figure 10. Colliding Block Relationship



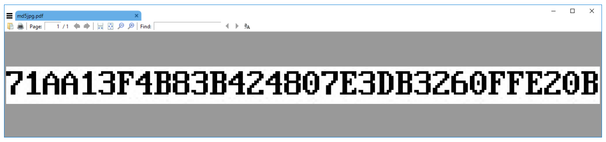
$ hexdump -vC shattered-1.pdf
$ hexdump -vC shattered-2.pdf
PDF Header
JPEG Start
JPEG Comment
Fixed
Comment length 0x173
Comment length 0x17F
Collision blocks
This is the only part of the files which is different
JPEG Comment
Desync
JPEG parsing gets out of sync here
Variable
JFIF Header
Quantization table
Quantization table
SOF2 header
Interleaving
Small comment on the right hides the header between the two large comments on the left
Real JPEG data starts much later...
JPEG Comment
Huffman tables
JPEG Comment
Image data
CC BY 4.0 Hector Martin 2017

Figure 11. How the SHA-1 collision PDF format trick works

comment skipping past image 0. The long-comment version instead sees the contents of image 0 followed by another JPEG comment extending right to the end of the image, whose size we'll hardcode for convenience. This lets us switch between image 0 and the other images without changing the MD5, and we repeat this process for images 1, 2, etc. The final image for `F` is displayed if no other image was selected, giving a total of fifteen collisions, repeated for each of the thirty-two digits.



```
C>md5sum md5jpg.pdf
71aa13f4b83b424807e3db3260ffe20b *md5jpg.pdf
```



71AA13F4B83B424807E3DB3260FFE20B

Since this doesn't require any clever PDF tricks the file[36] should work for any PDF, and because the image sizes are fixed in advance it could just have fixed-size placeholder images that are overwritten by the collision. Total running time is approximately an hour.

Alternatively, the PDF format has a feature called **`Form XObjects`**, effectively embedded mini-PDFs which can be displayed using "/objectname Do" and can be nested. If we can keep characters not allowed in a name out of the MD5 collision we can switch which XObjects get drawn and display the MD5 as actual text. (Thankfully enough PDFs draw text one character at a time that everything

[36]`unzip pocorgtfo14.pdf md5jpg.pdf`

handles this cleanly.) `block[15]` is as unconstrained as `14` and can become the `Do` command, meeting the (mostly irrelevant) length limit on names in PDFs, and avoiding most character restrictions on the second collision block. This turns out to save quite a bit of hacking time and runtime.

Of course, then we have to deal with implementation-specific fixes like disguising the trailing garbage as a string because `PDF.js` gives up otherwise, banning `0x80` and `0xff` which PDFium considers whitespace for some reason, and matching parentheses to properly terminate the dummy strings and keep Adobe Reader happy — but not counting escaped parentheses, or we'll add too many closing parentheses and break `PDF.js` again.

That's a lot of extra effort just to make copy-and-paste and `pdftotext` work, with no guarantee future software won't break it. It works though.[37]

```
$ pdftotext -q md5text.pdf -
66DA5E07C0FD4C921679A65931FF8393


$ md5sum md5text.pdf
66da5e07c0fd4c921679a65931ff8393  md5text.pdf
```

– – – –   – – –   – – – –   – – –   – – –   – – –   –   – – – –   – – –

How we put the MD5 on the Front Cover

a short addendum by Philippe Teuwen

On page 56, you'll see that this issue is a NES ROM polyglot that, when run, prints its own MD5 checksum. It would have been be a pity to not take advantage of the trick presented by Mako to get this very issue displaying the same MD5 on its cover page.

This required some productization of Mako's PoC, moving from a stand-alone Python script that creates a PDF from scratch to something that can be integrated with our existing LaTeX toolchain.

PdfTeX provides `\pdfximage` as a mechanism for embedding graphic objects, which, combined with `\immediate`, allows us to inject the sixteen JPEG tiles at the beginning of the PDF, right after the pseudo object containing the bulk of the NES ROM. This mechanism is accessed by means of `\pdflastximage` and `\pdfrefximage` wherever we want to use the injected tiles:

```
\immediate\pdfximage width 4.8pt {supertile.jpg}
\edef\mdfivetileAA{\kern 1pt \pdfrefximage\the\pdflastximage}
\immediate\pdfximage width 4.8pt {supertile.jpg}
\edef\mdfivetileAB{\kern 1pt \pdfrefximage\the\pdflastximage}
...
\edef\mdfive{\mdfivetileAA{}\mdfivetileAB{}...}
```

New tiles have been created to mimic the default LaTeX `monospace` font under the constraint that they, with the extra colliding blocks, can fit under a single JPEG comment, i.e. a total size fitting in a 16-bit word and *in fine* an average of 3,500 bytes per tile. Alternatively, it would have been possible to include higher resolution tiles, at the cost of crafting chained comment blocks.

To get both NES and title page MD5 right, the operations have to be properly interleaved: compile LaTeX sources with the `\pdfximage` objects; integrate the ZIP; insert a first PDF object with the NES ROM; insert the ROM header in front of the PDF header; compute the collisions for the ROM; insert a first set of collisions in the ROM; compute the collisions for the PDF/JPEG tiles; insert a first set of collisions in the PDF/JPEG tiles; compute the complete file MD5; swap collisions in the ROM; swap collisions in the PDF/JPEG tiles.

As we like to see the correct MD5 while typesetting without having to recompute the collisions systematically, we use two caches of the collisions that need to be renewed only if the MD5 of the prefixes change. With a little luck, that's only when the NES ROM or the JPEG tiles are modified.

Finally, we manually backport the collisions displaying the computed MD5 into the monoglot and inanimate PDF version of the issue provided to the print shop.

[37] `unzip pocorgtfo14.pdf md5text.pdf`