22:05 Inside Out; or, Abusing archive file formats.

by Ange Albertini

We have previously demonstrated hash collisions in documents with blocks of 64 bytes, such as the great MD5 pileup in PoC||GTFO 19:05. This used colliding, aligned blocks in `pocorgtfo19.pdf` to match a hash of `pocorgtfo19.exe`, `pocorgtfo19.png` and `pocorgtfo19.mp4`. That is to say, these files were not identical, but they did share an MD5 hash.

This research started with an incorrect assumption that Zip, TAR, and GZIP couldn't be generically exploited with collisions. Even with the almighty chosen-prefix collisions, I thought that Zips *may* not work, XML will *never* work, and GZIP will *always* trigger a warning.

Zip is the most collision unfriendly of standard file formats: bottom-up, pointers everywhere, duplicated data... Since they are officially parsed bottom-up, you can't even use a Chosen Prefix Collision on a pair of Zip files if their size difference is bigger than 64 kb, as the EoCD (end of Central Directory record) of the smaller archive will be too far from the end of the file to be found, thus making the file invalid.

On top of that, some critical data (such as file length, name, and content CRC32) is duplicated in the Local File Headers and in the Central Directory for a given file, which means it is present before and after the file contents—thus preventing any generic exploitation.

And unlike most archive formats, Zip is a tree of pointers between structures instead of sequences, so any size change of file content will propagate on the rest of the file: the last structure of the file contains a pointer and the number of archived files.

XML files also don't play nice with collisions: `CDATA` comments are defined in XML files, but they have to use the defined encoding, which is incompatible with the randomness of collision blocks.

XML files don't tolerate appended data either. It's another totally collision-unfriendly format.

DOCX files are Zip archives containing XML files and various data files, such as JPEG and PNG images.

[10] `unzip pocorgtfo22.pdf APPNOTE-1.0.txt`

[11] `git clone https://github.com/resilar/crchack.git`

Root file In DOCX files, the `/_rels/.rels` file plays a very special role. It's the *root* of the document, which points to other XML files of the document. It defines the *relationships* between the files.

You can move the files around provided you update the root, which requires a hard-coded path and filename in the archive. You can also make *two documents co-exist* in the same archive, pointing to either in the root file. A valid strategy to generically collide two documents seems possible.

Collision blocks You can't store the collision blocks after the XML content, since that would invalidate the root file's XML structure. And we can't easily forge the CRC of collision blocks, so we can't store them in the contents of a dummy file.

However, we can store the collision blocks in the Extra Field of a file, since Extra Fields don't have a CRC. Extra Fields were defined in 1990, in the very first version of the specifications.[10] They are commonly used and very extensible, so many implementations both ignore this field and preserve it. Extra Fields are stored before file contents, so they can't be stored in the Local File Header of the root; a dummy file stored after the root file can be used as a host for them.

It's easy to force the same length for the root file. We just need to choose two close paths for each document. Storing them rather than compressing them guarantees the lengths to be identical and predictable.

CRC You need to keep the root file CRC constant despite the collision blocks, since the CRC is duplicated near the end of the file in the Central Directory.

Forging a CRC is easy, but CRCHack makes it super easy![11] Just specify the bits you want, and it instantly gives you the requested output with the requested CRC32 without any encoding violation.

As an example, we now demonstrate forging a CRC with ASCII characters.

```
$ cat ascii
```

```
<!--ABCDEF-->
$ crchack \
  -b 4.0:+.8*6:1 -b 4.1:+.8*6:1 \
  -b 4.2:+.8*6:1 -b 4.3:+.8*6:1 \
  -b 4.4:+.8*6:1 -b 4.5:+.8*5:1 \
   ascii 0xdeadf00d
<!--tuI_\Y-->
```

Only with the uppercase bit of letters:

```
$ cat letters
<!--THESEKINDSOFCRCAREVERYIMPRESSIVE-->
$ crchack -b 4.5:+.8*32:.8 letters 0xcafebabe
<!--thEsEKIndsOFcRcAReVEryiMPREssIVe-->
```

So now we have two versions of the root files, with the same CRC, the same length, and via a dummy file with Extra Field containing HashClash collision blocks: the two Local File Headers that give the archive the same MD5.[12]

Results Unlike most reusable generic collision prefixes with a header and no body, this actually gives us two reusable generic collision pre-archives that are totally valid and manipulatable with standard tools. Provided you're careful with timestamps—either ignoring them in the source files or recompiling within two seconds—doing the same operations on both pre-archives will maintain the equality of hash values of both files, which is nice and very unusual.

Even better, deleting any archived files beside the root and the dummy collision block file will revert to the original hash values without any further modification required! Who would have expected that standard Zip tools could give you predictable hash values?

```
$ md5sum docx*zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx1.zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx2.zip
$ zip -oXll docx1.zip zinsider.py
  adding: zinsider.py (deflated 64%)
$ zip -oXll docx2.zip zinsider.py
  adding: zinsider.py (deflated 64%)
$ md5sum docx*zip
d12044feee801ad0530a911fa7f18db5 *docx1.zip
d12044feee801ad0530a911fa7f18db5 *docx2.zip
$ zip -d docx1.zip zinsider.py
deleting: zinsider.py
$ zip -d docx2.zip zinsider.py
deleting: zinsider.py
$ md5sum docx*zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx1.zip
6c33d52590ff0bb0cc8cdafe6aa5153b *docx2.zip
```

Supported formats This trick is applicable to any file format made of a Zip-ed XML with a root file. It works for `.docx`, `.pptx`, and `.xlsx` from Office, for the open container format in ePub, and for other open packaging conventions, such as `.3mf` for 3D manufacturing and the XML Paper Specification, `.xps` and `.oxps`.

Corkami collisions' `zInsider` makes it possible to instantly collide any of these formats, with precomputed prefix archives.[13]

This is easy to extend to any other similar format, but a new prefix pair must be recomputed for any new format.

Some formats like Quake's PK3 aren't exploitable: they don't have a root file to abuse. The Open Document Format requires their root file to mention every other file, which isn't generic. APK, JAR, and XPI are even worse: they require all the other files' hashes!

Gzip

TAR files have no room for any abuse: pure sequences of headers with hardcoded size and offsets, then file contents. No declared lengths, no skippable content. You can use chosen-prefix collisions on them, but that's it: nothing generic.

Gzip doesn't seem to be playing nice with hash collisions either: any extra data is placed before the compressed file contents, and appended data typically triggers a warning and is not taken into account for parsing anyway. Gzip collisions are possible, but not in a generic way.

However, while most Gzip files start with the typical `1F D8` structure—called a *member*—it's actually specified that a Gzip file can contain several of these members, in which case the data of each will be decompressed and concatenated. So a member with no compressed data but with extra data acts as a comment that can be parasitized, albeit quite a complex one.

Since the length of the Extra Field is stored on two bytes in little-endian before the Extra Field itself, it's even exploitable with UniColl![14]

[12] `git clone https://github.com/cr-marcstevens/hashclash`
[13] `git clone https://github.com/corkami/collisions; find collisions -name zinsider.py`
[14] `git clone https://github.com/corkami/collisions; find collisions -name unicoll.md`

So a generic reusable hash collision for Gzip is actually possible via a classic sequence of comments. First one comment to align the rest of the file to collision block boundaries, then one comment whose length is variable—its encoded value will be overlapping with one of the differences in the collision blocks—and then we start two chains of comments to toggle one payload or the other, exactly like we did for JPEG, MD5, or SHA1.

Colliding GZIPs like JPEGs Like JPEG, we have this limit that extra field can't be bigger than 64 kb, but recompressing data in chunks of 64 kb is much easier with Deflate than with JPEG! Since the decompressed data of all members is concatenated, we just need to cut the archived data in chunks.

This idea isn't new. Some formats like BGZIP (2008) chunk the data in several members and store an index in the extra field, making it easier to decompress some contents separately while maintaining a standard Gunzip-compatible structure. This is a common source of multi-member Gzip files.

So it gives us reusable hash collisions for anything that relies on Gzip as outer encryption, such as `.tar.gz` or SVGZ. As long as the data is decompressed, the structure of the outer archive can be freely modified.

However, some programs like Inkscape use their own lightweight implementation of Gzip, which doesn't support files made of several members, so our collision strategy will not work in these exceptional cases.

Conclusion

While Zip, XML, GZIP, and TAR seemed very hostile to collisions, combining several tricks made it possible to get generic reusable hash collisions for GZip archives (`.tar.gz`) and Zipped XML files with a root, such as DOCX files.

The strategies are very different, even if they both rely on the extensible Extra Field which is similar in both formats. For DOCX, it's a merge of two documents inside the same Zip, with two versions of the same root file. For Gzip archives, Extra Fields are used as comments, and two independent archives are interleaved via two chains of skip and data.

Other formats aren't playing that nice: Bzip2 is a pure compressor, bit-based with only bit alignment, and no padding and no form of comments. Other formats such as XZ, AR or Compress (`.Z` archives) are just too simple for any exploitation. RAR applies CRC16 to headers, which does not help our cause.

Thanks for Yann Droneaud for the TAR.GZ challenge, and Philippe Lagadec for the DOCX challenge!

Still using MD5? It might feel useless to still care about MD5, but as MD5, SHA1, and SHA2 use the same construct, exploits of hash collisions via file format tricks will be re-usable for other hash collisions while being cheaper to pull off with MD5. These techniques would work for SHA1 via the Shamble attack too, except that it costs $45,000 USD to compute it. And at least, MD5 is still widespread enough that it has enough targets to attack in practice, unlike MD2 and MD4!

You might be tempted to still use MD5 to designate a file, but using MD5 will expose you to all kinds of tricks and confusion that SHA2 or Blake2 don't.

Fastcolls are very quick to compute and can be chained, providing one bit of stored data while keeping the MD5 constant.[15] They will make it trivial to watermark a file, and a very short shellcode can easily detect which version of the file is running, then adjust its behavior accordingly. Using a stronger algorithm would prevent any possible pranks or confusion, at least for some years until we get better collisions.

Bonus: ZGIP Zip can use Deflate among other compression algorithms. On the other hand, Gzip only uses Deflate.

Both are wrapping Deflate data around different structures that are not compatible. By abusing structures, it's possible to make ZGIP, a chimera of Zip/GZIP: a polyglot file sharing the compressed data.[16]

By abusing Deflate stored blocks and dummy members, it's even possible to partially hide some data from the other format, even if they belong to the same stream.

In short, this is just going the extra mile to prove that GZIP is not a wrapper around Zip, nor Zip is a wrapper around GZIP.[17]

[15] `git clone https://github.com/brimstone/fastcoll`
[16] `git clone https://github.com/corkami/pocs; find pocs -name zgip`
[17] `https://speakerdeck.com/ange/gzip-equals-zip-equals-zlib-equals-deflate`

Bonus: The craziest colliding file The latest advanced MD5 manipulation is a very clever ZStandard+Tar hashquine+polyglot by David 'Retr0id' Buchanan,[18] also known for his beautiful PNG hashquine.[19]

It can either be just a Zst file, but also a Tar.zst, so the Tar header can be toggled on or off, as well as the complete tar checksum. To be a reusable hashquine, it's able to output any MD5 and Tar checksum while keeping the whole file's MD5 constant.

The same prefix is reusable in three different ways. First it can be a pure ZStandard file hashquine.

```
$ md5sum hashquine.zst
720ca7f6842f1a608fcb924f5811ebb9 *hashquine.zst

$ zstd -cd hashquine.zst
The MD5 of hashquine.zst is:
720ca7f6842f1a608fcb924f5811ebb9
```

Second, it can be a Zstandard(tar) file.

```
$ md5sum hashquine.tar.zst
703911cf9e409965cebd05392acc1503 *hashquine.tar.zst

$ tar -Oxf hashquine.tar.zst hash.md5
The MD5 of hashquine.tar.zst is:
703911cf9e409965cebd05392acc1503
```

Finally, it can be a self-checked "auto-manifest" Tar.zst.

```
$ md5sum self.tar.zst
f068d54fabb12dbb1b359745a80d78fc *self.tar.zst
~
$ tar -xvf self.tar.zst
x hash.md5
x hello.txt

$ cat hash.md5
f068d54fabb12dbb1b359745a80d78fc *self.tar.zst
ed076287532e86365e841e92bfc50d8c *hello.txt

$ md5sum -c hash.md5
self.tar.zst: OK
hello.txt: OK
```

The whole prefix uses 653 Unicolls to toggle Zstandard frames and output optional contents after decompression.

For the optional Tar Header (generic for any `hash.md5` contents), it uses one frame for the constant Tar header start, 8∗11 frames for the `hash.md5` file size in octal, one frame for the constant Tar timestamp `14412572240`, 8∗6 frames for any tar header checksum in octal, and one frame for the rest of the tar header.

For the optional text prefixes in the file contents, it uses one frame for the constant prefix of "The MD5 of hashquine.tar.zst is" and other for "The MD5 of hashquine.zst is" in ASCII. Finally, it uses 32∗16 collisions for all nybble possibilities of an MD5 hash.

Bonus: Wordpad weird files Colliding `.docx` files will show the same document with Microsoft Wordpad. It turns out that Wordpad ignores the root files entirely, and just locates the document file via the Content Types files. Really!

As you would expect with such sloppiness, it doesn't check if all files in the archive are declared in the Content Types file, which can turn any Zip archive into a very weird `.docx` that is Wordpad-only with just two XML files. Sadly, this issue being far from a standard file. Wordpad is confused as it should be, and we can't make this issue a Wordpad-compatible DocX file too. Extract an example from this PDF's attachments.[20]

[18] `git clone https://github.com/corkami/collisions; find collisions -name hashquines`
[19] `unzip pocorgtfo22.pdf retr0id.zip; unzip retr0id.zip hashquine_by_retr0id.png`
[20] `unzip pocorgtfo22.pdf mini.docx`