This Dump Is A Puzzle

reconstructing j2me firmware from an unknown file system

nccgroup

Overview

- A little bit of back story
- An interesting problem appearing
- The path I took to solving it
- Some upper bounds on efficacy of my approach

Introductions

- Security consultant at NCC Group for the last 2.5 years
- Prior hobbyist background
- Heavily multiclassed hacker
  - Appsec
  - Hardware
  - Amateur Radio
  - Some crypto/maths
  - Some RevEng (obviously)

Once upon a time

- Investigating an IoT gateway device.
- Chipset based on running J2 Micro Edition midlets on some proprietary JVM/OS.
- Device bridges LPWAN -> GSM/Private APN
- Usual boring unsecured debug serial pwnage.
- Obviously can’t just stop there!

Warning: Cursed Image!

- Have an eSIM on the board

Warning: Cursed Image!

- Have an eSIM on the board
- Wanted: direct APN access for back end systems!

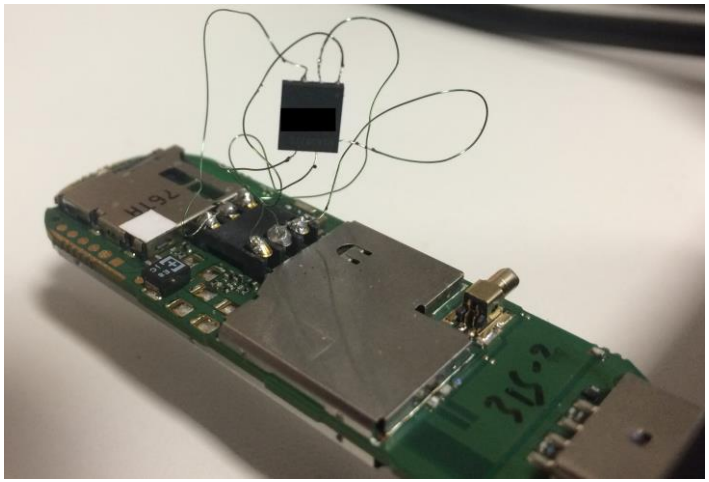
nccgroup

Warning: Cursed Image!

- Have an eSIM on the board
- Wanted: direct APN access for back end systems!
- Desoldered the eSIM

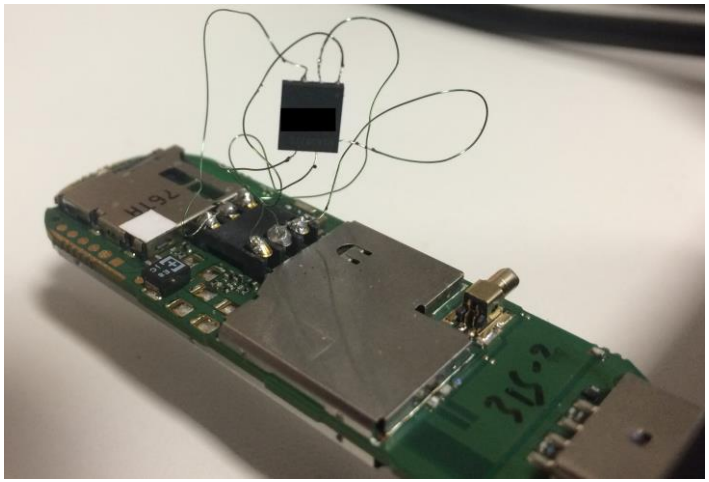
Warning: Cursed Image!

- Have an eSIM on the board
- Wanted: direct APN access for back end systems!
- Desoldered the eSIM
- Super janky kludge onto a cheap mobile GSM dongle
  - Three prayers of contrition to the gods of EMC compatibility.
  - I'm genuinely very sorry.



Cursed image!

- Have an eSIM on the board
- Wanted: direct APN access for back end systems!
- Desoldered the eSIM
- Super janky kludge onto a cheap mobile GSM modem
  - Three prayers of contrition to the gods of EMC compatibility.
  - I’m genuinely very sorry.
- Still need APN config/creds to connect



nccgroup

Hunting For Creds

- Abusing the pwned java environment?

Hunting For Creds

- ~~Abusing the pwned java environment?~~
- Extract the eSIM data?

Hunting For Creds

- ~~Abusing the pwned java environment?~~
- ~~Extract the eSIM data?~~
- Used a logic analyser with an ad hoc IEC 7816 parser to MITM bus traffic between the SIM and the board and see what turned up

Hunting For Creds

- ~~Abusing the pwned java environment?~~
- ~~Extract the eSIM data?~~
- ~~Used a logic analyser with an ad hoc IEC 7816 parser to MITM bus traffic between the SIM and the board and see what turned up~~
- Dump the chip and dig through that?

Hunting For Creds

- ~~Abusing the pwned java environment?~~
- ~~Extract the eSIM data?~~
- ~~Used a logic analyser with an ad hoc IEC 7816 parser to MITM bus traffic between the SIM and the board and see what turned up~~
- Dump the chip and dig through that? Sure!
  - Can see fragmented file chunks
  - Including Zip file fragments – found our J2ME data!

Chip dump investigation

- They can’t simply be extracted!
- Can see unidentifiable proprietary file system
- Data all looks like regular-sized pages
- Can’t find a page table anywhere
  - Or anything that looks even like offsets from the start of the file data

Demo 1

Just binwalking it

Chip dump investigation

Which leads to the inspiration for this piece of research:

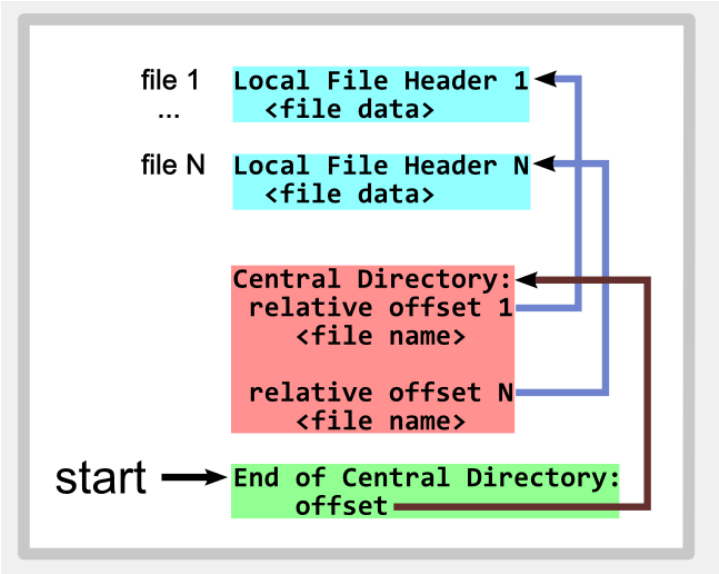
- Zip files have a lot of internal structure/metadata/self-reference
- Maybe I don’t have to care that I’ve got no page table
- Treat the dump as a set of shuffled puzzle pieces
- Find clues in the metadata to match pieces to their appropriate locations.
- Why not write a puzzle solver (a completely absurd shotgun parser) to help reorder the pieces into coherent data?

Fragmented Decompression

- Compression algorithms (including DEFLATE) use tightly packed bitstreams
- Not completely structureless
- However: This structure is mostly incoherent without context
  - It's difficult to tell whether any given set of bits are a bit-packed fragment header, a byte code for a stored pattern to expand, or part of a literal data string.
  - Decompression depends on the compression state machine being in the right state for each bit being processed, which also depends on previous bits processed.
- Using Binwalk to do generic "unpack DEFLATE streams" results in completely irrational output on fragmented/disordered data
  - In my case, a very small dump unpacks unboundedly, eating up all the spare disk space (many gigabytes!) before crashing, while losing the associated zip file metadata
- TLDR: Decompression requires that the compressed blob is intact.
  - Or at least, considerably increased complexity determining sufficient context to recover it - e.g.: http://blog.ptsecurity.com/2017/12/huffman-tables-intel-me.html

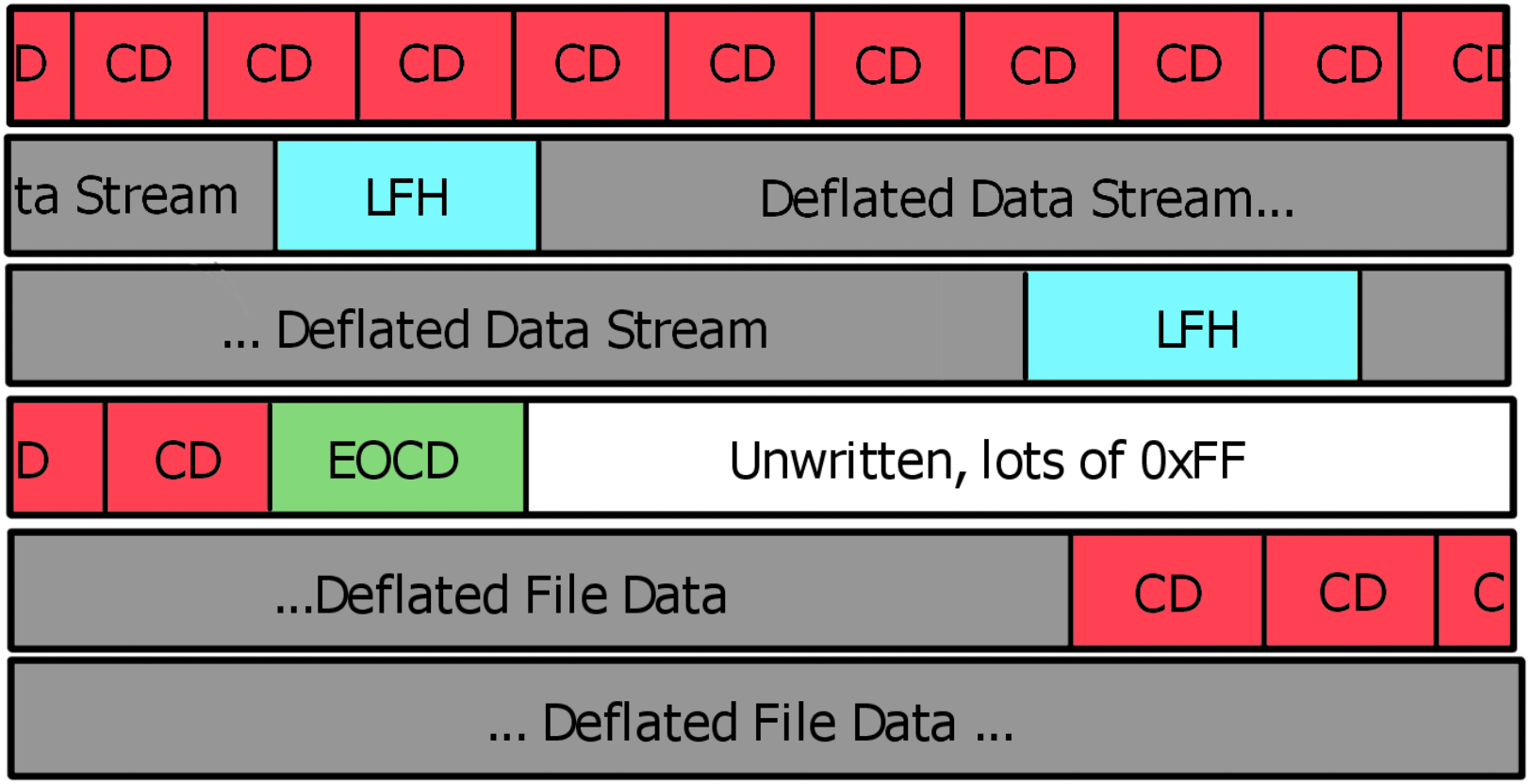
Brief intro to Zip files

- Thanks to Ange Albertini's Corkami project producing diagrams which helped a lot with the intuition for this!
- Zip files exhibit a three level hierarchy.
- End of CD header points to the Central Directory
- Central directory points to Local File Headers
- LFHs are directly followed by associated data



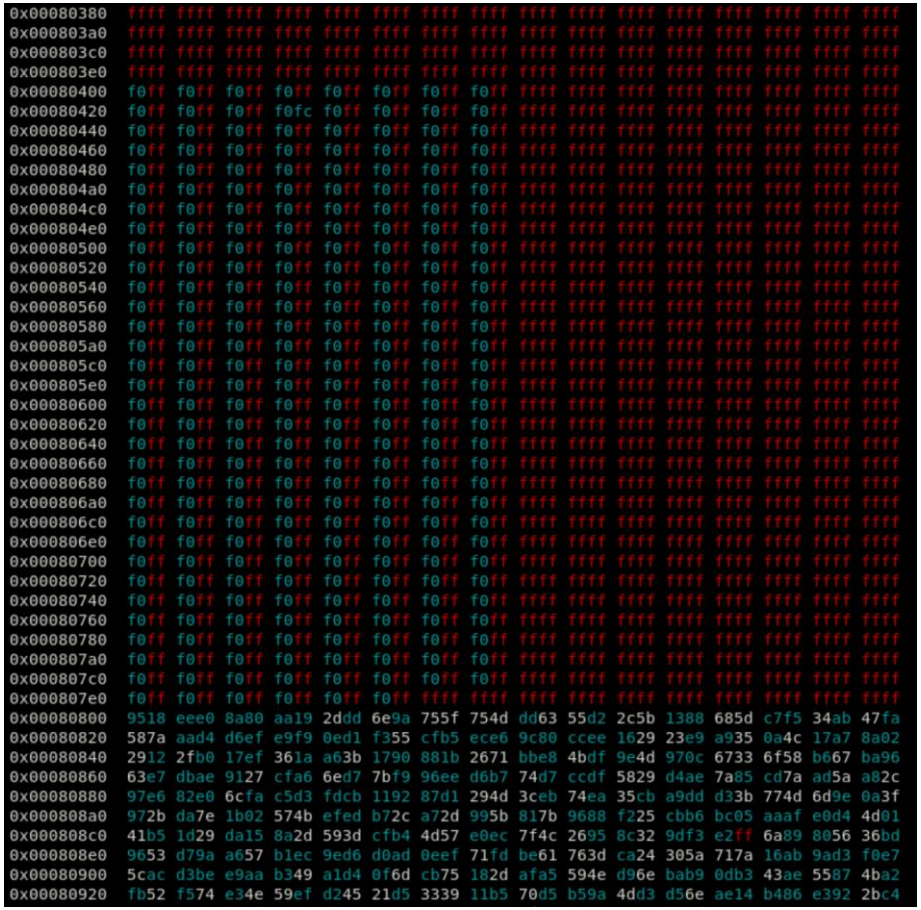
nccgroup

Corruption


D
CD
CD
CD
CD
CD
CD
CD
CD
CD
CD
CD
ta Stream
LFH
Deflated Data Stream...
... Deflated Data Stream
LFH
D
CD
EOCD
Unwritten, lots of 0xFF
...Deflated File Data
CD
CD
C
... Deflated File Data ...

Characterising a dump

- Skim through dump
- Page boundaries (and multiples of that are fairly clear at a glance in a hex editor.
- Good idea of how large our “puzzle pieces” are
- Can use this as the basic unit for reconstructing our files



Where do we start?

End of Central Directory gives us:

- The final record in a zip file
  - Happens only once!
- Total size of file headers + data
- Total number of file entries to expect
- Size of Central Directory
- Easily identifiable magic signature `PK\x05\x06`

```
End of central dir record:

  end of central dir signature    4 bytes  (0x06054b50)
  number of this disk             2 bytes
  number of the disk with the
  start of the central directory  2 bytes
  total number of entries in
  the central dir on this disk    2 bytes
  total number of entries in
  the central dir                 2 bytes
  size of the central directory   4 bytes
  offset of start of central
  directory with respect to
  the starting disk number        4 bytes
  zipfile comment length          2 bytes
  zipfile comment (variable size)
```

nccgroup

First complication

```
[0x00080760]> / PK\x05\x06
Searching 4 bytes in [0x0-0xb00000]
hits: 2
0x0076c549 hit1_0 .kon.classPK\u0005\u0006m.
0x00a9d4b6 hit1_1 .:zz.classPK\u0005\u0006ff3.
[0x00080760]>
```

Two instances in the dump

- Pointer data in zip files is only usable if we can tell which one it's from
- Need to be able to classify fragments before reordering them
- Reading the header data indicates ~ 2000 entries between them
  - Problematic number of entries to be reviewing manually for classification.
- Trying to solve this puzzle solving problem as generally as possible (not just pick out classifiers for one particular dump)

Automating classification

Looking at potential classification criteria in the headers that we can use to sort the records into a collection for each expected file:

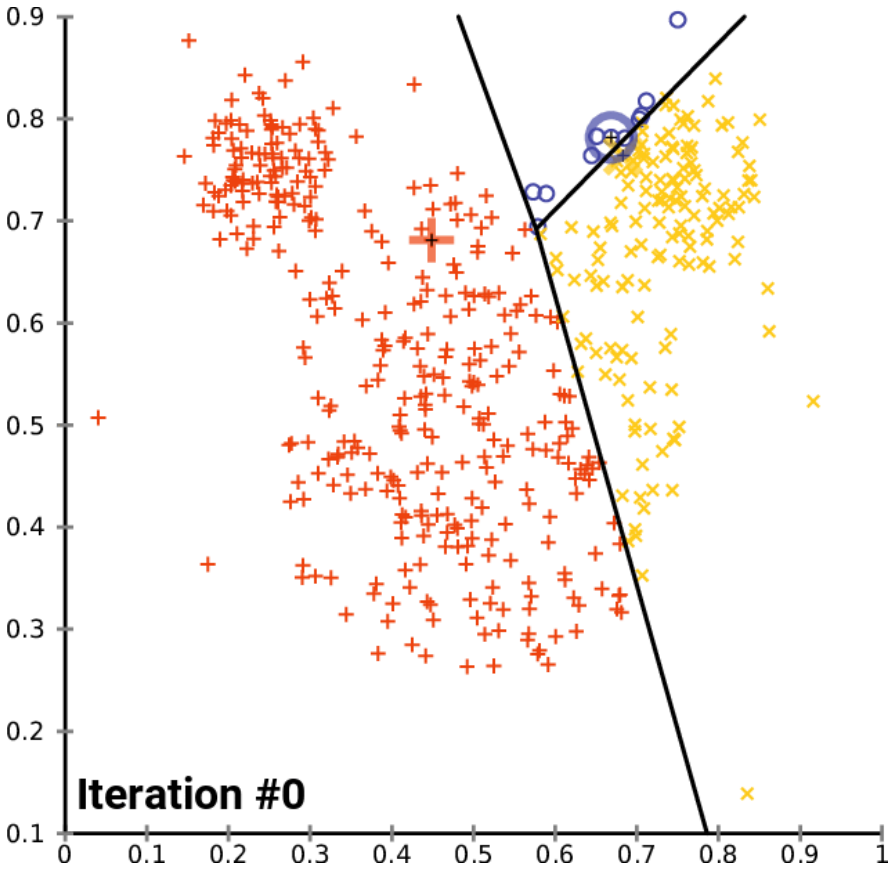
- Version data (OS and software-stack indicators)
- Compression flags
- Compression method
- Timestamp (presuming midlets not compiled simultaneously!)

```
File header:

  central file header signature   4 bytes  (0x02014b50)
  version made by                 2 bytes
  version needed to extract       2 bytes
  general purpose bit flag        2 bytes
  compression method              2 bytes
  last mod file time              2 bytes
  last mod file date              2 bytes
  crc-32                          4 bytes
  compressed size                 4 bytes
  uncompressed size               4 bytes
  file name length                2 bytes
  extra field length              2 bytes
  file comment length             2 bytes
  disk number start               2 bytes
  internal file attributes        2 bytes
  external file attributes        4 bytes
  relative offset of local header 4 bytes

  file name (variable size)
  extra field (variable size)
  file comment (variable size)
```

Applying k-Means clustering

- Find and parse all available CD headers
- Convert the version/flags/method/timestamp fields into separate fields of an “observation” vector.
- Spread *k* markers randomly among distinct observations,
- Then move them iteratively until they each sit in the centre of clusters.
- Rough informal description of the algorithm:
  1. Assign all observations to their nearest marker
  2. Move the marker to the average of all its assigned observations.
  3. While marker movement > some small amount, GOTO 1

k-means clustering



Animation from https://commons.wikimedia.org/wiki/User:Chire

(our use case is 5-Dimensional and difficult to visualise usefully)

nccgroup

So now we've got our Central Directories

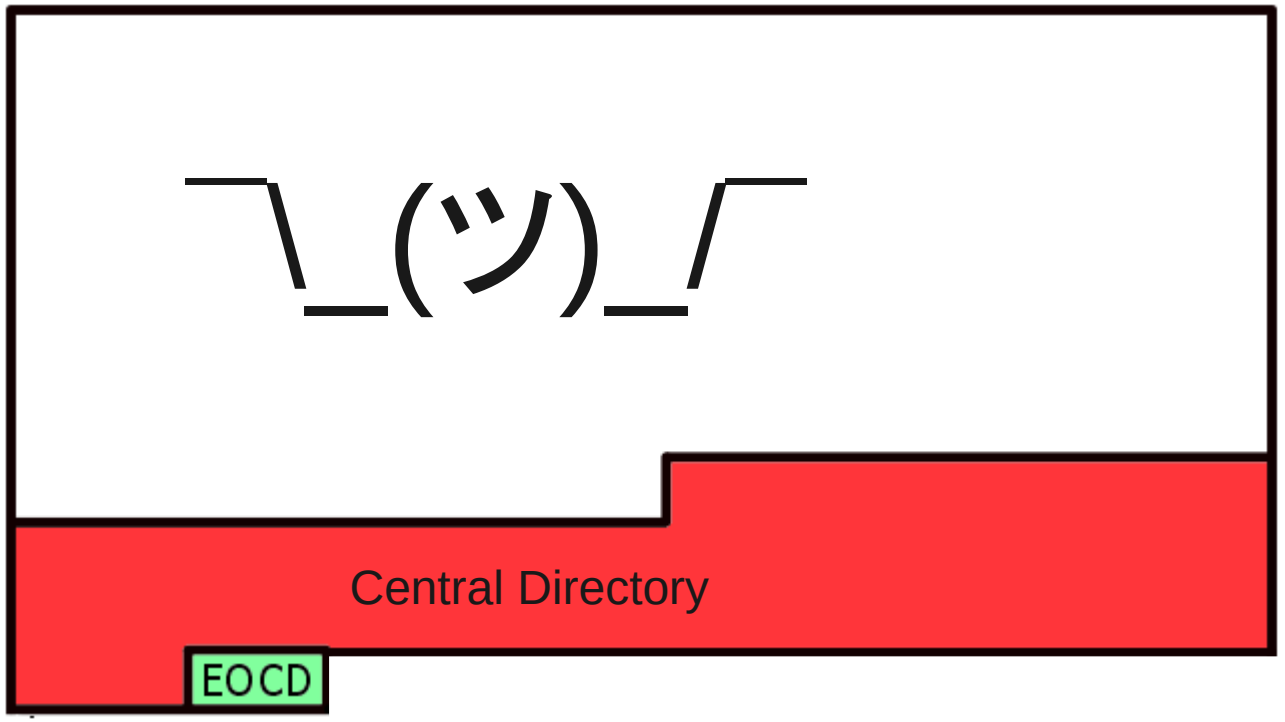
- Use the LFH relative offset for ordering.
- Each record generally created sequentially as the file is being built up
- Create a skeleton file for each zip file
- Start with moving the page for the first CD to where the EOCD indicates it should be
- Carry on for remaining CD pages finishing with the EOCD page.

```
File header:

  central file header signature   4 bytes  (0x02014b50)
  version made by                 2 bytes
  version needed to extract       2 bytes
  general purpose bit flag        2 bytes
  compression method              2 bytes
  last mod file time              2 bytes
  last mod file date              2 bytes
  crc-32                          4 bytes
  compressed size                 4 bytes
  uncompressed size               4 bytes
  file name length                2 bytes
  extra field length              2 bytes
  file comment length             2 bytes
  disk number start               2 bytes
  internal file attributes        2 bytes
  external file attributes        4 bytes
  relative offset of local header 4 bytes

  file name (variable size)
  extra field (variable size)
  file comment (variable size)
```

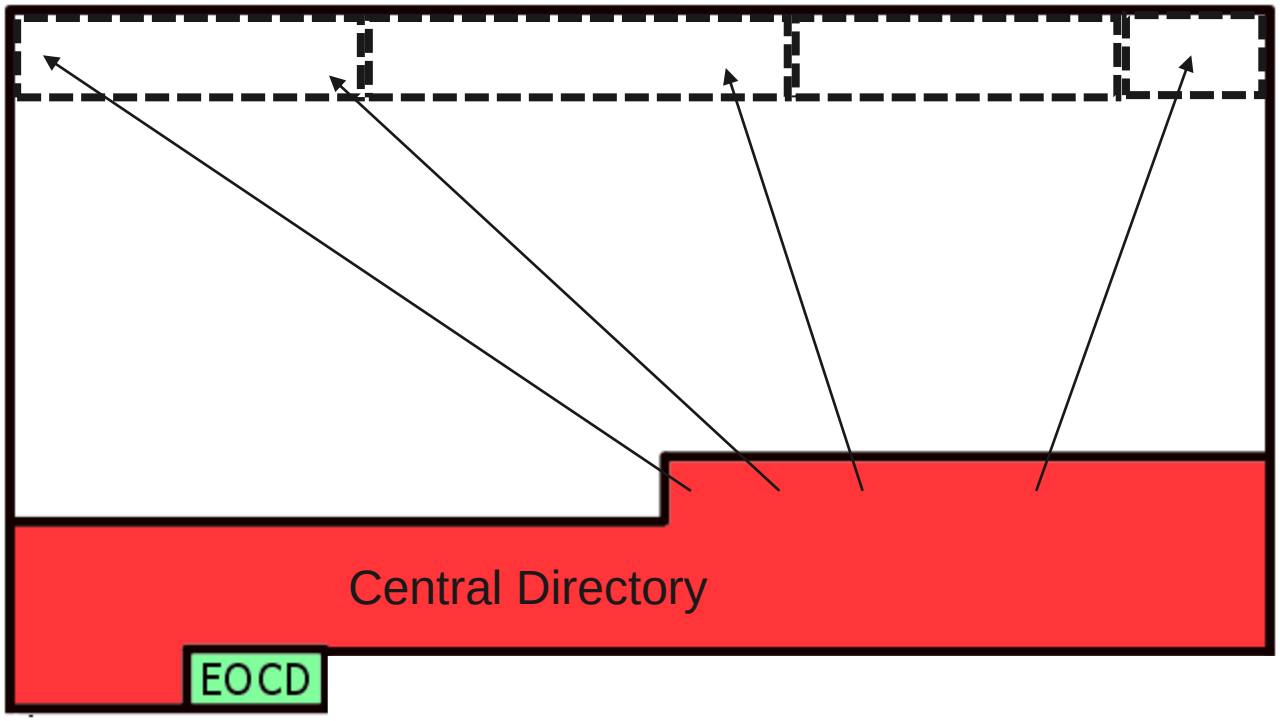
nccgroup

Where we are so far

- Any broken CD headers which spanned page boundaries are now recovered
- CD headers are densely packed and give a complete page ordering for central directory.

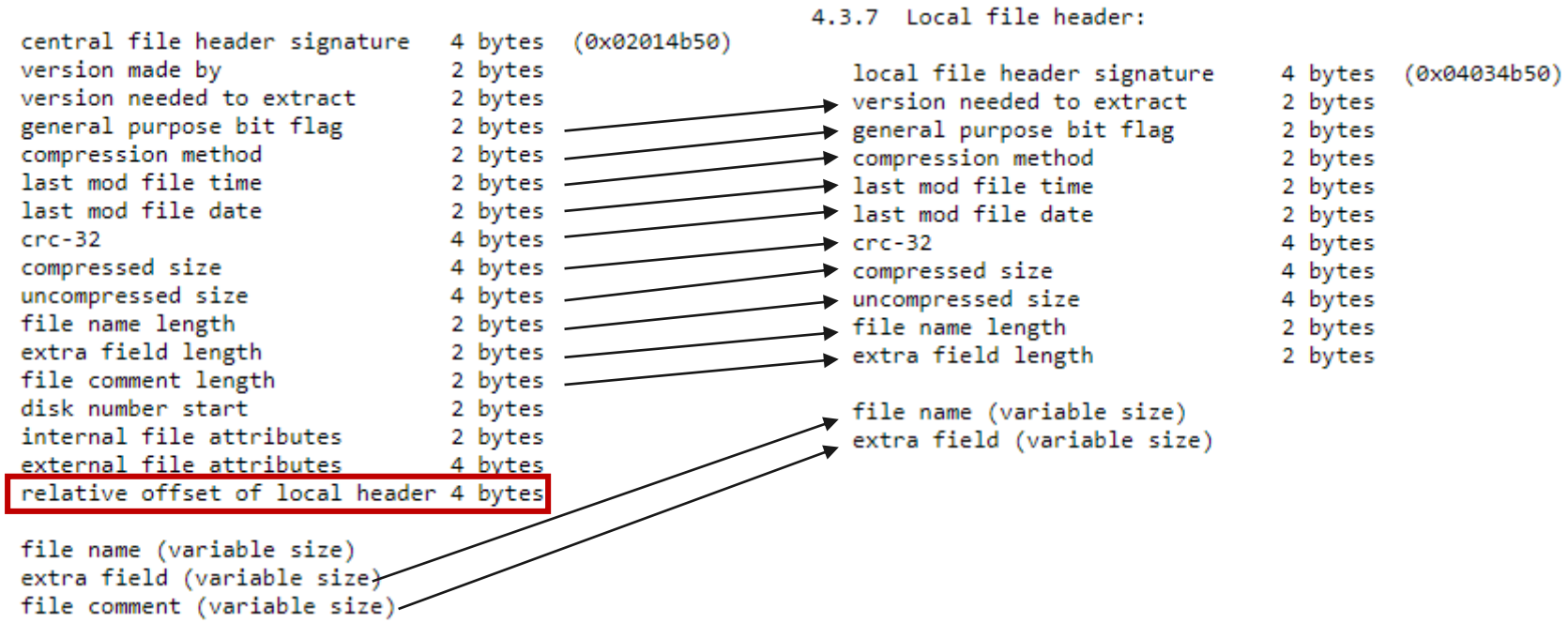


Building a file map

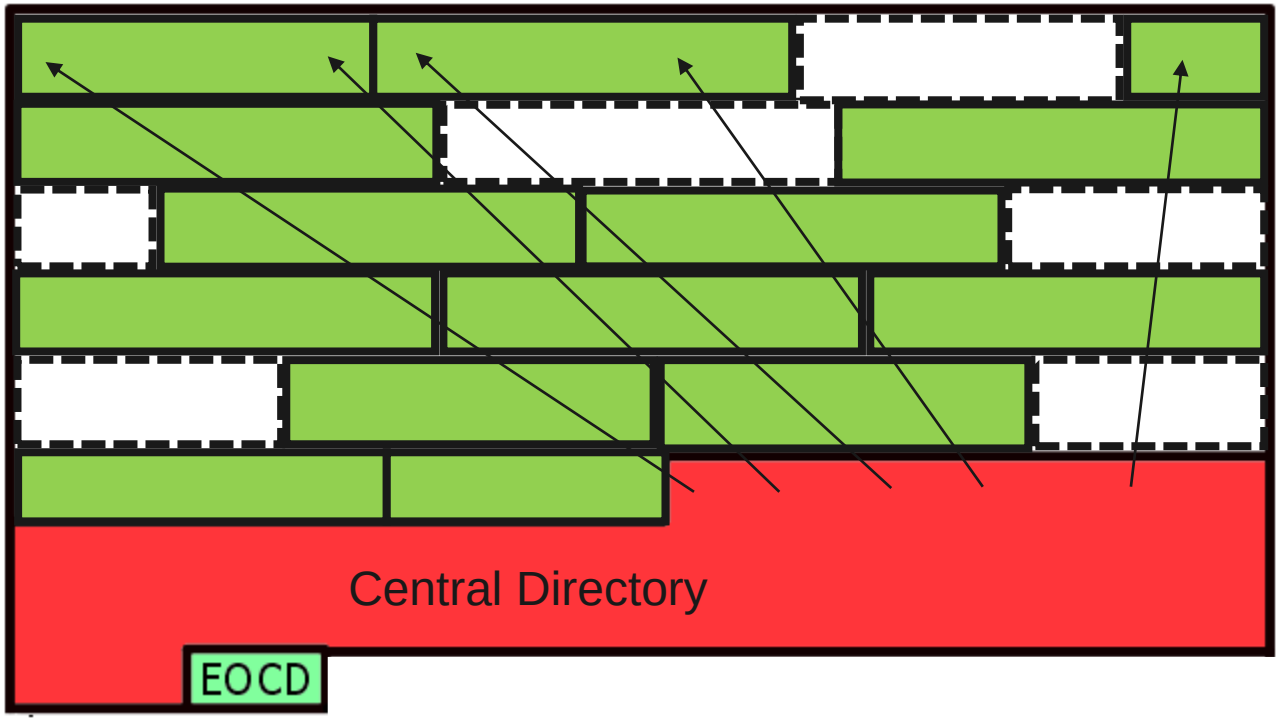


Finding matching file records

From the PKZip Specification



Known pages filled in



Demo 2

Ziprecover POC script in action

(Have actually re-implemented the whole thing in Rust to try and leverage the well known nom parser macro crate for convenient parser verification but introduced an unsolved perf bug recently which makes total runtime too long for demo comfort. Sorry)

Limitations

- Can’t fully recover any data segment bigger than twice page size (although thanks to the header data, up to the first 2 pages of compressed data can be unzipped nicely!).
- Worst case where LF headers bookend a page of pure compression data, lose a full file chunk (but will retrieve file data for the files before and after)
- I haven’t made use of the CRC data yet to try and search for matching pages to fill gaps.
  - This would work pretty well for filling one-page gaps
  - However, combinatorics means even though CRC is cheap, testing it over many permutations of a file gets exponentially hard quickly for multi-page gaps, increasingly likely to see a CRC32 collision instead of the correct permutation of pages.
- Also still some implementation bugs
  - This is a massive shotgun parser and I’m not doing enough sanity checking during header parsing to prevent corruption artefacts creeping in and breaking it in all cases.
  - There are a lot of ways to get parsing of corrupted data wrong!

Thanks for listening.

Any Questions?

nccgroup