22:03 Mitra and Mocky: Near-polyglots and Mocks

by Ange Albertini

Our readers are encouraged to read Abusing File Formats (PoC‖GTFO 7:6). This is a follow-up with better classification, leading to the creation of tools to automate the generation of various kinds of weird files: mocks, polyglots and near-polyglots. We also share a tool, Mitra, to help apply these techniques.

The basic idea to abuse files is to make space for foreign data that will be ignored by parsers. This can rely on various features of the targeted file format.

Signature Some formats like MP4 and PostScript are parsed from offset zero, but they don't enforce a magic at that offset. As a consequence, it's possible to make a MP4 / PostScript polyglot by abusing the length of an MP4 *atom* to encode a PostScript declaration for a dummy function that will cover more MP4 structure.

A polyglot MP4/PS header starting a dummy function but also an MP4 *free* block.

```
    /PostScript/ whitespace then line comment
00:00 00 00  %  f  r  e  e \r \n  %  !  P  S \r \n
   \----MP4---\ Declares a comment of length 0x25

   /PostSc./ Declares a function name
               then a line comment
10: / {   (  % 00 00 00 00 00 00 00 00 00 00 00 00
20:00 00 00 00 00 00 00 XX  f  r  e  e \r \n  )  }
               \---MP4----\ Declares a comment len
```

The same bytes look quite different when interpreted as a PostScript file, where `00 00 00 XX` declares the length of a free chunk covering the whole PostScript.

```
\0\0\0%free
%!PS
/{(\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\xXXfree
)}
```

Starting parsing at offset zero is only a half-measure against file-format abuse. Only when signatures are enforced at the very beginning will file format abuse and confusion become impossible.

On the other hand, having a signature for each frame such as Ogg or ILDA can be superfluous if the frames are properly length-defined.

Cavities Some formats start with *cavities*, which are totally ignored. We used to call these empty spaces.

A typical example is the ISO format, being a raw dump starting with empty sectors. This is commonly used in IsoHybrid to make a single ISO that boots either as a thumbdrive or as a CD-ROM.

Another example is the DICOM format, which by courtesy skips the first 128 bytes of the file. A magic signature and enforced structure are required after that range. This is particularly convenient for TIFF files, since unlike most picture formats, the eight-byte long Image File Header points to the File Directory, which points to the image data, so it's easy to move the big chunks of structure and data around the DICOM ones. In short, a TIFF/DICOM polyglot is easy, and a TIFF/DICOM chimera—sharing the same image data—is even possible.

Appended data Most parsers will ignore any further data once the file is considered "complete enough," which is determined by whether some kind of recovery is required or not.

It might be explicit via some kind of terminating marker. Or it might not, when enough data is present or by intentionally triggering a parsing bug to end the parsing. An example of triggering a recursion exhaustion warning can be found in PoC‖GTFO 17's AGC polyglot.

Data added after the end—and ignored—is called appended data, but it's rarely mentioned explicitly in the specifications. It's simply tolerated.

Some formats actively prevent appended data by having a footer. A footer can be thought of as a specific structure required to be at then the very end of the file such as ID3v1, which is a footer appended itself to a MP3/Layer3 stream, or the XZ archive format, which officially enforces a footer to define a file as complete.[6]

Other formats like Dicom, Ogg, ILDA or PCAP[NG] enforce the whole file to follow a given structure: they're just pure sequences, sometimes even requiring a magic signature for each structure such as ILDA or Ogg. They just go on parsing everything from the file, and returns an error since it's invalid.

[6] See "The .xz File Format, v1.0.4."

Metadata Since it's present in the file but not needed for parsing or rendering the file's contents, metadata is a great source of abuse. Many old school formats have fixed length fields, as hard coding was the norm back then.

Comments are typically ignored empty space with typically a set length that is declared before, and are present in most file formats. Unlike PDF, XML enforces an encoding for its comments like the rest of the file, but in general, comments are just ignored, and preserved, no matter their amount, their length or their content.

Comments are not the only source of abuse. Extensible metadata with a user-chosen ID, or fields like file names in an archive can also be used to store some foreign data. A notable exception is that in Gzip, the optional comment and file name are null-terminated, which shows that they're intended to store standard text, while the also optional Extra Field is defined with a 2-byte length.

As a side note, metadata may seem like a permanent risk entirely, and it's natural to wonder why we define them officially in every format if they are so easily abused later. While metadata doesn't seem like an initial requirement to keep the format simple — like the Quite OK Image format — it is eventually needed to be able to keep extra information in the file, which is exactly what happened for the MP3 files.

At the release of *l3enc* (the original Mpeg Layer 3 encoder in 1994), the files initially had an `l3` extension, had no file format whatsoever. They were pure sequences of layer 3 frames, each with their own frame header with no signature, making them hard to detect and easy to confuse with other data such as JPEG segments.

Since there was no way to store any metadata in L3 files, the compatible ID3v1 *footer* with a hard coded length was unofficially defined. More structures were defined in other clumsy ways around the L3 stream (Xing, Lame, APE, ...), showing the need for proper definition of metadata storage from the beginning. ID3v2 eventually defined a header, a magic which gave at last a proper format to MP3 streams.

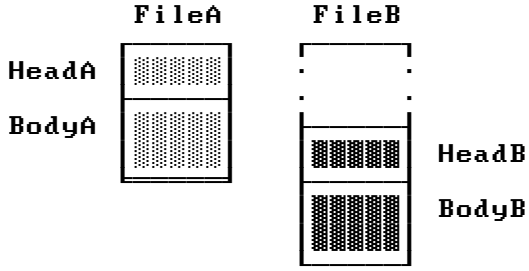
It's a shortsighted move to come up with a great compression algorithm (e.g., MP3, QOI) and define a way to store some data in a file format without the ability to extend it with new but optional data in the future. Even if it means that these structures can be abused, you can't have an extensible format that can't be abused, and people *will* extend or fork your format if not, requiring an extra format that could have been avoided from the beginning.

Wrappending Some formats don't tolerate appended data, but they can end with a parasite-hosting structure, which acts like appended data, just wrapped in a declarative structure. Since most data-storing structures have all their declaration before their data itself, it behaves like appended data from the outside, even if the length of the appended data has to be declared somewhere. It looks like appended data — ignored but tolerated but it's technically an appended parasite — declared and skipped.

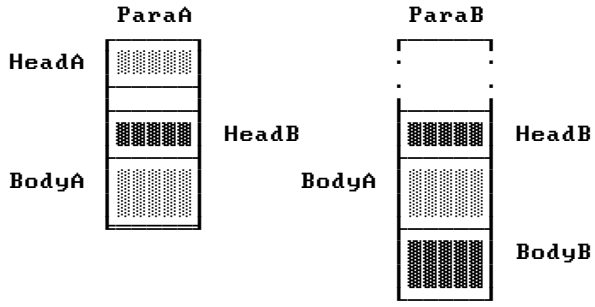
Zipper Some formats have very strong constraints: a tiny cavity, or very small parasite length (256 bytes for a GIF). Rather than parasitizing a whole file, let's just add the declaration of the file, a declaration of a comment, and then store the rest of the file as appended data.

A zipper is a fabric construct of two sliders where each tooth interlocks with the other side's tooth. As an analogy, a *zipper* is a file construct where each format comments the other format's elements, and each tooth is a parasite for the other format.

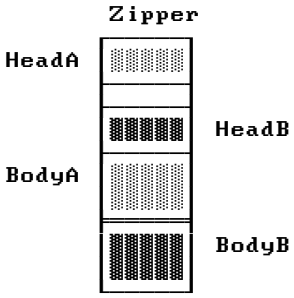
The simplest form of a zipper is made of two formats: Format A starts at offset zero, tolerates appended or wrappended data, and Format B starts with a cavity. Both formats can be parasitized.



We parasitize File A with Head B, adding padding if required. We also parasitize File B with Body A, *wrappending* Body B in advance if required.



When we merge these files, it looks like this.



Zippers combines various format features (cavity, parasite, appended data) to overcome limitations and make even more weird formats combine.

`Mitra` is a tool that combines all this knowledge for 40+ different format, generating hundreds of format combinations with different strategies.[7]

Mitra is a simple tool. It doesn't understand file formats structure, it just contains the minimum amount of information to identify and parasitize a file format. It expects standard files as input!

Abuses

Payload embedding The simplest form of exploitation is to just embed a payload that doesn't need to be a valid file. In this case, use the `--force` command line parameter.

The universal example for that is HTML or JavaScript that can be embedded in most file formats. If the file is too big, the HTML page might take too long to load entirely. In that case, use JavaScript to break out of the appended data and limit the parsing to the web payload only.

[7] `git clone https://github.com/corkami/mitra`

[8] `unzip pocorgtfo22.pdf mocky.py`

Mocks While 7:6 covered file type identification, it didn't cover any exploitation. The easiest way to exploit file type identification is just to give a binary blob the right signature at the right offset. It could even happen accidentally. Such a file is a *mock* file. A simple example is `FF D8`, a two byte file that can be identified as a `JPEG image`.

Polymocks It's also interesting to just add a mock signature at a given offset in a valid file via any of the previously mentioned techniques.

`Mocky` is a tool that uses the Mitra library to insert specific filetype signatures at specific offsets to create polymocks.[8] Since programs like `file` have so many signatures that they are scanned by alphabetic order of their category, it's possible to predict which detection will be returned first, and the order might not be what you'd expect from a threat model perspective.

Of course, it's possible to cram as many signatures as possible within the constraint of the target format, such as this issue, a valid PDF with many extra mock signatures stored in a standard stream object. Here is an example of such a *polymock* file, with many filetype detections yet no valid content:

Running `file` with `--keepreading` gives an impressive list of detected formats:

```
Plot84 plotting file
SymbOS executable v7.z, name: ...
Old EZD Electron Density Map
Zoo archive data, vj., modify: v78.88+
Symbian installation file
Scream Tracker Sample adlib drum stereo...
Poly Tracker PTM Module Title: "MZ...
SoundFX Module sound file
Nintendo Wii disc image: "NXSB...
DICOM medical imaging data
Linux kernel ARM boot executable...
VirtualBox Disk Image, minor 8653 (MZ...
JPEG 2000 image
ARJ archive data
COM executable for DOS
unicos (cray) executable
data
```

Mocky has a `--combine` flag for to try and insert as many signatures in a file as possible.

`file` has an extra weakness that it has special support for tar files before any other format, and can identify tar files not by their magic, but by the validity of their header checksum, even without any tar signature in the file. This is used in this issue too, so even if the file is a standard PDF starting with a generic PDF signature, `file` with no parameter sees it as a `tar archive`, even if it doesn't contain a magic Tar signature.

Mocky will adjust the tar checksum if used for a polymock file. Here is such a empty `mock.tar` file, detected generically as `tar archive` even if it contains no signature at all:

```
000: 00000000 00000000 00000000 00000000
...
090: 00000000  4 0 0       00   00000000
...
1F0: 00000000 00000000 00000000 00000000
```

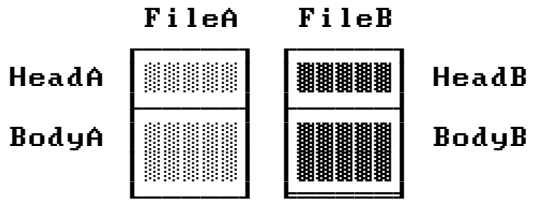
Adding a valid tar checksum to the previous polymock example will indeed return a tar filetype — if the `--keep reading` parameter isn't used — despite all the other present signatures.

Near-polyglots Formats that require a different signature at the same offset can't be combined in a polyglot. However their combination can still be exploited in different conditions. *Near polyglots* are files that are almost polyglots, except that some bytes have to be replaced so that the file type changes.

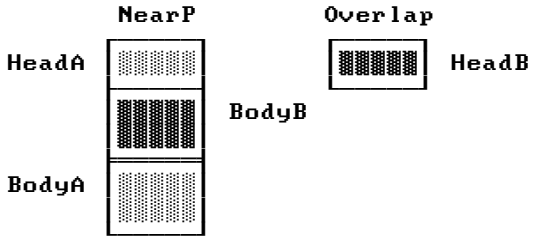
This change could happen over the network if some packets arrive in a different order. It could also happen due to weak bits, leading to different contents. And it can happen via a cryptographic operation in which case you can call them *crypto-polyglots*.

One of these use case is Angecryption, introduced in PoC||GTFO 3:11, where I demonstrated abusing the Initialization Vector of CBC, CFB or OFB block modes to replace the first block of the crypto-polyglot. A new abuse of pseudo-polyglots is presented in the TimeCryption article on page 30 of this release.

To generate a near polyglot, you need very light constraints. Format_A can be parasitized and Format_B can start at the same offset, and needs to tolerate appended or wrappended data. Technically, generating a near polyglot is like parasitizing a format, ignoring that they both start at overlapping offsets, and keeping the head.



Just parasitize File_A with Body_B and keep Head_B as the overlap:



The minimum length overlap is basically the number of bytes where you can declare a file then have some unparsed space, either naturally or by declaring a comment. This value can change drastically between file formats, as shown on page 15.

For example, it's 1 for PostScript because you can declare a line comment with %, so provided there's no encoded newline *after* decryption, this will be a valid ignored space. The PE file format's minimum overlap is two bytes — `M Z` — because you can abuse the DOS header, limiting you to a 58-byte parasite.

A JPEG header with comment declaration is `FF D8 FF FE XX YY`, which is six bytes, with `XXYY` being the length of the comment in big endian. However, if you need a `0x3489`-long comment, a `0x35??`-long comment will do the trick, so you don't have to bruteforce the `YY` byte. If you feel luck, you might

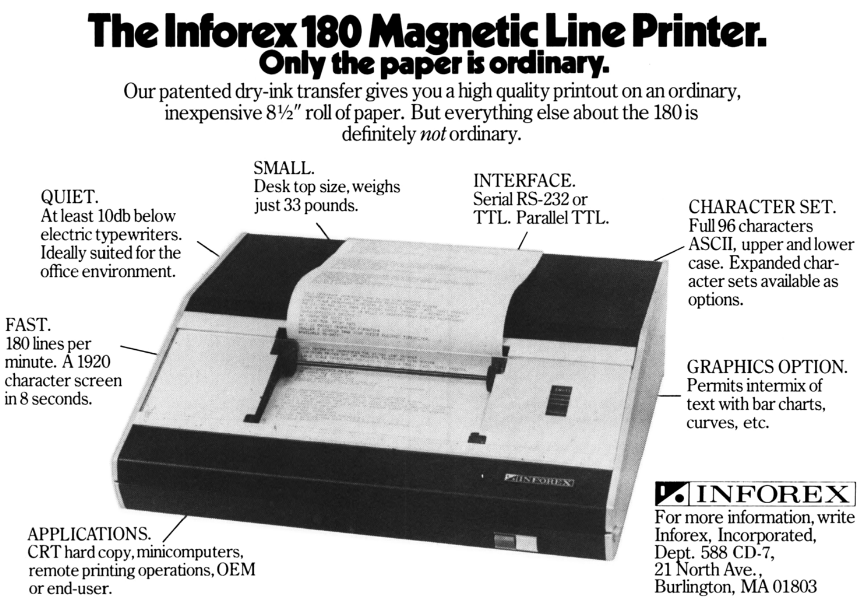
```
                                                               Variable  Unsupported
                                                                offset     parasite
Minimal start offset
         1 2 4 8     9  16  20  23  28  34  40  64  94  132    12  28
                      12          26  32  36      68  112 226    16

         P P J F M T F W G P R I R B C I P C J P E A P I I J    W B O      B E G L N
         S E P 1 P I L A Z N I D T M P L S A P C L R C C C a    A P G      Z B I N E
             G a 4 F V D   G F 3 F P I D D B 2 A F   A O C v    S G G      2 M F K S
               c   F         F v     O A       P     P     a    M            L
                               2               N
                                               G

1* PS    . M A ? ? ? ? ? ? A ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?    ? ? ?      ? ? ? ? ?
2@ PE    M . A A A A A A A A A A A A A A A A A A ! ! ! ! ! !    M M M      ! ! ! ! !
4+ JPG   A A . A A A A A A A A A A A A A A A A A A A A A A A    A A A      A A A A A
.  .
.  .     [the table could go on but would take too long to bruteforce]

           X: automated     ?: likely possible
           M: manual        !: unknown
```

* Hack that relies on line comments with GhostScript. Requires the parasite not to contain any new line, after encryption.

@ Hack relying on overwriting the DOS Header, therefore restricting the parasite space to offsets 2-60.

+ Signature, comment declaration and length are two bytes apiece. To specify them all is six bytes, but if we round up the big-endian length and leave its low byte uncontrolled, we only need five. And if we leave the length entirely uncontrolled, we only need to fix four bytes.

Figure 3: Minimal start offsets of file formats, and exploitation via near polyglots



also gamble on the length and not bother to brute-force `XX` either.

Mitra can generate such files with the `--overlap` parameter. It keeps the overlap's content in the filename as well as the offsets where the content changes formats, to be re-used later by AngeCryption or TimeCryption scripts.

Ambiguity Files with different interpretations depending on parsers we now call *ambiguous* and previously called *schizophrenic*, werewolves or shapeshifters. There are plenty of sources of this ambiguity.

When a value such as a pointer never changes across standard files, it's tempting for a parser to simply ignore it. Putting some contents under unusual conditions while putting other contents under the typical conditions might reveal a difference between the two parsers.

Sometimes a value is represented twice. For example, a buffer with a declared length might also end with a null terminator. What if that terminator happens earlier than the declared length? Which length value is the real one? Or if you declare the same value twice, and there isn't an error, does the first or the second declaration take priority?

If you corrupt a format on purpose and the parser tries to rebuild the file, how does it do it first? What if you put a valid file structure in a comment? Such recovery algorithms are typically not officially specified, so each developer might do it differently.

Some formats are extending older formats. Both the old and the new formats are present in the file. These formats are naturally ambiguous at a format level, and we might call them ambiguous polyglots.

A widespread example is the Portable Executable, defined as an extension of the DOS format. Preciously few PE files—such as `regedit95.exe`—have a meaningful DOS payload. Valid PE files are expected to just have the same DOS stub with no unique code.

Robert Xiao proved us wrong by crafting a universal Doom binary, which works from DOS 6 to Windows 10, both as valid DOS and PE payloads in the same file.[9] This is something like a father and son having the same name, with no distinctive suffix whatsoever.

Ciphertexts can be ambiguous too, even despite authenticated encryption! Check the TimeCryption article page 30 in this issue for abuses of GCM, GCM-SIV and OCB3.

Collisions PoC‖GTFO 19:05 covered a lot of details for exploiting hash collisions. However tar.gz and DOCX (ZIPped XML)—which were initially thought to be unexploitable—are properly dealt with and explained in the Inside Out article on page 19.

Conclusion

With basic knowledge of file format identification and abuse, Mitra can try different strategies and generates many forms of file abuse: payload embedding, mock files, polyglots and pseudo-polyglots.

Pseudo-polyglots are the unified. form of file formats abuses to be combined with cryptographic operations. They include both AngeCryption, covering ECB, CBC, CFB, OFB modes, and TimeCryption, covering CTR, OFB, GCM, OCB3, GCM-SIV modes.

Extensions of Mitra might cover ambiguous files with standard strategies, hash collisions and hash collisions over different formats.

[9] `git clone https://github.com/nneonneo/universal-doom`