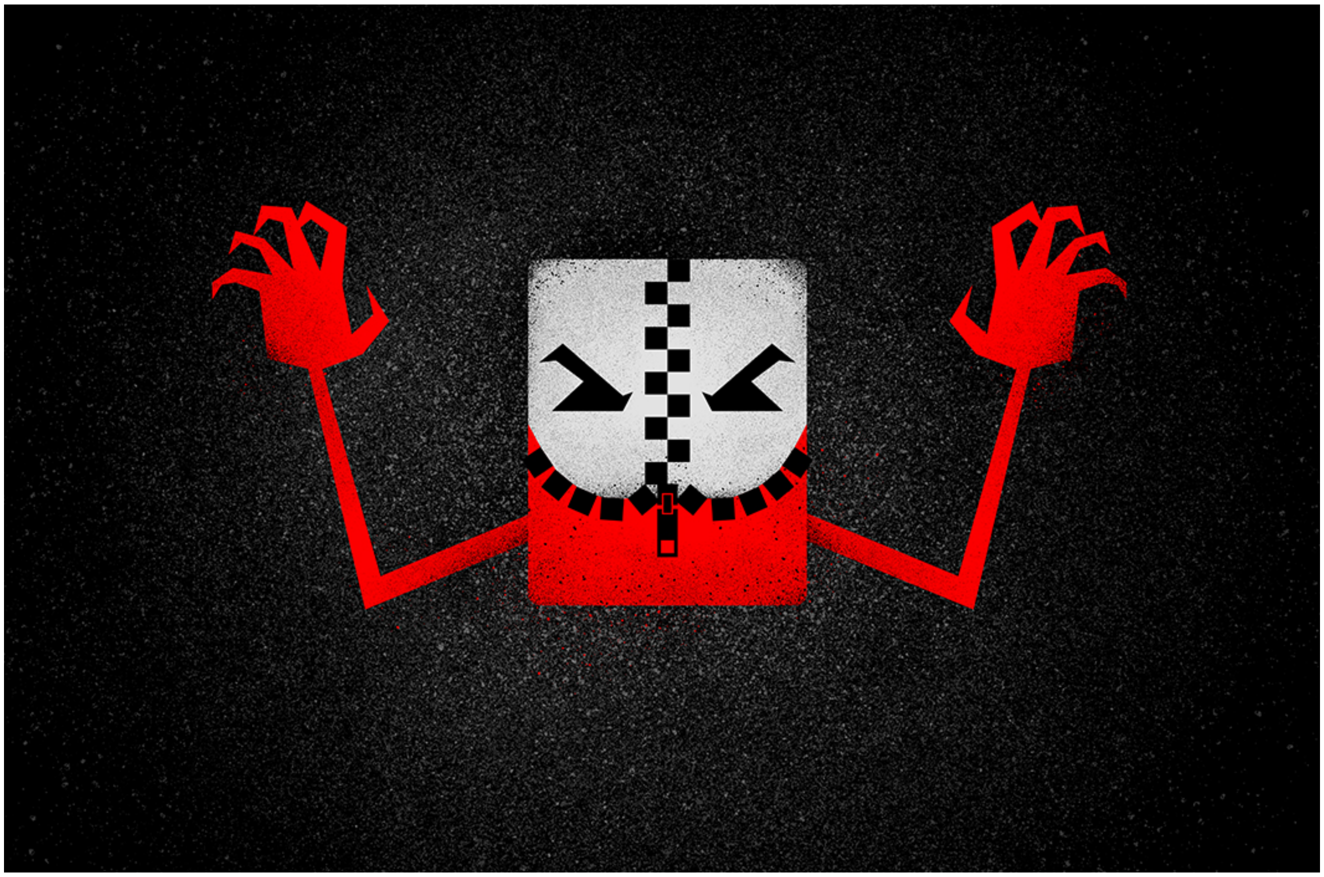
How to Prevent Zip File Exploitation

crowdstrike.com/blog/how-to-prevent-zip-file-exploitation/

Rich Seymour June 22, 2021



ZIP files are a known vector for phishing campaigns, ransomware and other malicious action. Because the format isn't generally executable (minus self-extracting ZIPs), it hasn't gotten as much attention as executable formats. This blog post looks at how the format can be exploited and shares the solution we came up with.

Compressed file formats come in many flavors such as tarballs (.tar.gz), RAR Archives (.rar) and 7Zip (.7z), but ZIP has become the foundation for widely used file formats in addition to becoming the generic term used for compressing and bundling files. It forms the basis of Microsoft OfficeOpenXML files (docx, xlsx, pptx file extensions), Java Archives (JAR), Android Packages (APK) and Electronic Publication (EPUB) files. ZIP structures are also found inside of self-extracting EXEs, hiding in PDFs and other obscure formats.

The redundancies inherent in the ZIP format, combined with Postel's Law (i.e., "be conservative in what you send, be liberal in what you accept"), have created a wide variety of "acceptable" ZIPs. Along with the variety, choices made by ZIP parsers can lead to a single ZIP producing different output based on the ZIP parser used. In this post, we highlight some of the redundancies and parts of the format that leave the interpretation of a given file in the hands of the application reading it. These differences in interpretation can be exploited by adversaries, and we'll hint at what we're doing to see these maliciously crafted ZIPs from all angles.

ZIP Name Confusion

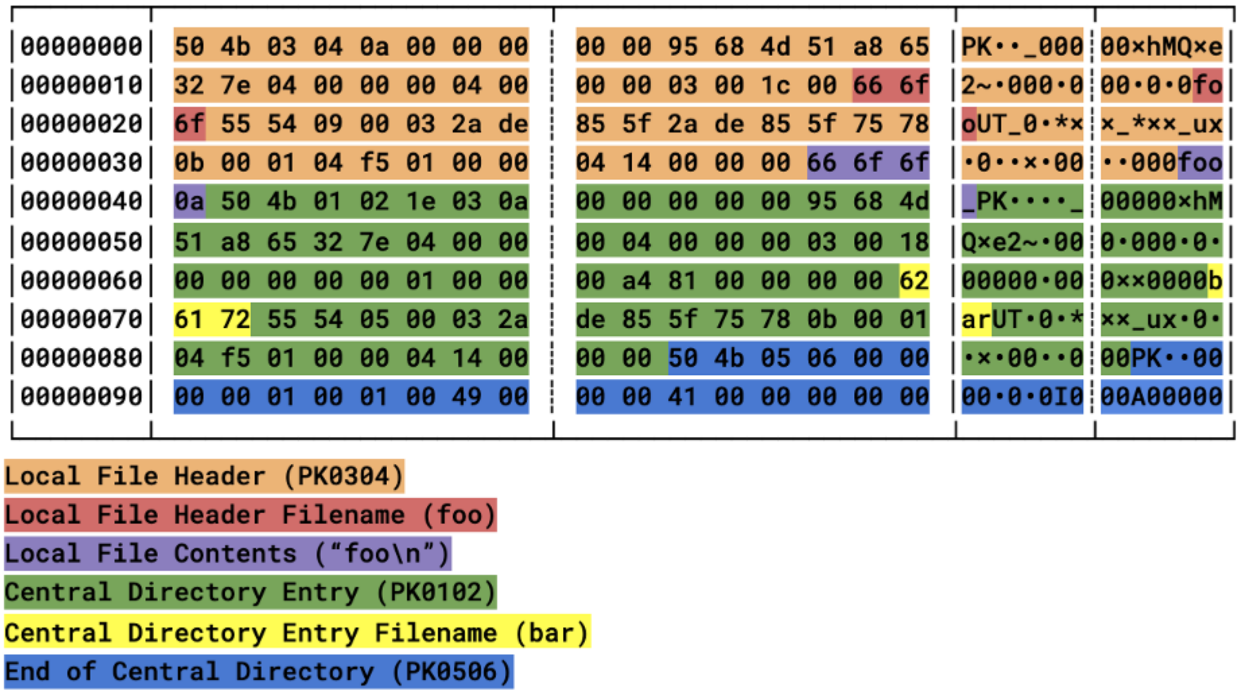
The ZIP format has been explained visually by corkami and others, but I will attempt to reduce the format for the sake of this example to two chunks of data that can provide a name for a stored file. Outside of symlinks and hard links, filesystems generally enforce uniqueness of file names. ZIP on the other hand does not.

If you glance at corkami's ZIP 101 poster, you can see that ZIP is meant to be read from the bottom up, unlike many other formats. But odd things can happen if you read in the other direction.

Two data structures in the ZIP format can be responsible for holding the file name, specifically the Central Directory Entry and the Local File Header. The Central Directory is at the end of the file (right before the End of Central Directory structure mentioned later), and a Local File Header is prefixed to each of the stored files, earlier in the file. Parsers can be confused by putting two different names into each structure that refer to the same data. The malicious impact is negligible (assuming a ZIP parser patched for the ZIP Slip vulnerability) but it provides a nice starting point to examine the complexities of how different parsers respond to malformed ZIP files.

A short example file and program can make this more concrete.

`centralbar-localfoo.zip`



```
|00000000| 50 4b 03 04 0a 00 00 00 ┊ 00 00 95 68 4d 51 a8 65 |PK••_000┊00×hMQ×e|
|00000010| 32 7e 04 00 00 00 04 00 ┊ 00 00 03 00 1c 00 66 6f |2~•000•0┊00•0•0fo|
|00000020| 6f 55 54 09 00 03 2a de ┊ 85 5f 2a de 85 5f 75 78 |oUT_0•*×┊×_*××_ux|
|00000030| 0b 00 01 04 f5 01 00 00 ┊ 04 14 00 00 00 66 6f 6f |•0••×•00┊••000foo|
|00000040| 0a 50 4b 01 02 1e 03 0a ┊ 00 00 00 00 00 95 68 4d |_PK••••_┊00000×hM|
|00000050| 51 a8 65 32 7e 04 00 00 ┊ 00 04 00 00 00 03 00 18 |Q×e2~•00┊0•000•0•|
|00000060| 00 00 00 00 00 01 00 00 ┊ 00 a4 81 00 00 00 00 62 |00000•00┊0××0000b|
|00000070| 61 72 55 54 05 00 03 2a ┊ de 85 5f 75 78 0b 00 01 |arUT•0•*┊××_ux•0•|
|00000080| 04 f5 01 00 00 04 14 00 ┊ 00 00 50 4b 05 06 00 00 |•×•00••0┊00PK••00|
|00000090| 00 00 01 00 01 00 49 00 ┊ 00 00 41 00 00 00 00 00 |00•0•0I0┊00A00000|
```

Local File Header (PK0304)
Local File Header Filename (foo)
Local File Contents (“foo\n”)
Central Directory Entry (PK0102)
Central Directory Entry Filename (bar)
End of Central Directory (PK0506)

Ruby code (adapted from: https://github.com/rubyzip/rubyzip/blob/master/samples/example.rb):

```
#!/usr/bin/env ruby
require 'zip'

## reads from localfile header name
Zip::InputStream.open('centralbar-localfoo.zip') do |zis|
  entry = zis.get_next_entry
  puts "#{entry.name}"
end

## reads from central directory name
zf = Zip::File.new('centralbar-localfoo.zip')
zf.each_with_index do |entry, index|
  puts "#{entry.name}"
end
```

This concisely demonstrates in one script how two functions that look very similar can produce different results. In the first, the ZIP is read in a streaming mode that finds files by the Local File Header alone, in the second it is read from the Central Directory Entry after the stored data is trusted. This has some interesting history in that System Enhancement Associates (SEA) ARC format, popular in the early floppy shareware days, just had file headers and no central directory. A lawsuit by SEA against Phil Katz for his PKARC utility led him to create ZIP. From looking at the original release docs, it seems that the central directory was helpful in keeping track of archives that spanned several floppy disks. Floppies have disappeared (except for the save icon) but ZIP has remained.

While this isn’t a threat vector per se (or even unexpected to experienced rubyists), the variations in ZIPs don’t stop at giving a file two possible names. File sizes, compression methods, duplicated names and concatenated zips can all give difficulties to programmers trying to open them safely.

Additional Methods

File Sizes

File sizes — the size of the compressed and uncompressed data — are stored in the Central Directory Entry or the Local File Header, but can also be stored in extensions to the ZIP format known as ZIP64, or in an structure known as the Data Descriptor, which acts like a footer to the stored data. Disagreements in the compressed size can lead to different extracted files, especially if the files are stored only (i.e., added to the ZIP uncompressed).

Compression Methods

Compression methods define what algorithm was used to compress the data stored in the ZIP, or none in the case of files too small to compress or stored uncompressed on purpose by the ZIP creator. They are an entire field in the aforementioned structs, but luckily only the DEFLATE method has widespread acceptance, and others such as LZMA are sometimes seen. Because DEFLATE blocks can also be uncompressed, the compression method tied with size and offset disagreements can allow subsets of files to be extracted by one parser but the full file extracted by another.

Duplicated Names

Similar to the foo / bar name confusion in the centralbar-localfoo.zip, specially crafted ZIP files can have names that collide with one another in a variety of ways. Multiple Local File Headers can simply have the same name, Central Directory Entries can report duplicate names, and there’s no protection from one overwriting the other if the unzipping application is blindly writing files.

Concatenated ZIPs

Concatenated ZIPs are one of the simplest to construct evasion methods, such as this nanocore delivery method. Based on the aforementioned issues, you can see how two concatenated ZIP files might look like a valid ZIP file to a parser, but which one gets extracted is up to the design. For all of these cases, we wanted a parser that wouldn’t make an opinionated decision but would give us every possible file that could be hiding inside a ZIP. One more example of ZIP confusion relating to concatenated ZIPs follows.

End of Central Directory Offset Confusion

The End of Central Directory (EOCD) is the suffix at the end of a well-formed ZIP file. It acts a bit like if the last page of a book told you which page the index (the Central Directory itself) was on. In books, we normally think of them starting with page one, and in binaries we similarly think of them starting from offset zero. But ZIP can be embedded within other file formats, meaning there can be different interpretations of what offset zero means. Different parsers can make different choices — Python's **zipfile**, Rust's **zip-rs** and Info-Zip (standard **unzip** command on Linux/Mac) take the offset to the Central Directory as an offset from the start of the file. Go's ZIP archive extraction starts looking from the start of the compressed data. This means you can have two different Central Directories within one file and see completely different contents depending on the interpretation of the same offset by a given unzipping tool.

In addition to file name and EOCD offset confusion, the ZIP format is robust (or weak) enough that many other tricks can fool ZIP parsers. If an adversary knows what tool is reading their ZIP payload, chances are they can craft a ZIP to confuse it. Or they can make the ZIP look enough like another file to evade detection. For instance, some threat actors will have their stager download what looks like an image file, but is actually ZIP files with the file fingerprint of the image file prepended or simply renamed.

Our Solution: Build an Omnivorous UnZIP

We could have used existing parsers in our language of choice (Rust) but even that parser makes choices when confronted with the confusing ZIP issues above. We wanted a tool that could explore and extract from ZIP files with all sorts of anomalous content, so we decided to make our own.

We decided to create a ZIP parser that doesn't make opinionated choices when confronted with disagreements in a ZIP file. Instead, we attempt to extract the superset of information from any given ZIP it is confronted with. In other words, if we get two options for a compressed size, we try them both. If we see two names for the same file, we report them both. If we see two files with the same name, we extract them both for processing.

Our preliminary results are promising, and we are continuing work to make sure our parser gets as many interpretations of ZIP files as programmers have seen fit to implement. In a future blog post we'll get into more specifics about our Robust ZIP Parsing in Rust. If there is community interest we may open source this library and the tools based on it for use by the InfoSec community and anyone interested in exploring the vast corner cases inside of this useful and popular file format.

Additional Resources

- *Read more from our Data Science team: Building on the Shoulders of Giants: Combining TensorFlow and Rust.*

- *Learn about recent intrusion trends, adversary tactics and highlights of notable intrusions in the CrowdStrike 2021 Threat Hunting Report.*
- *Learn more about the CrowdStrike Falcon® platform by visiting the product webpage.*
- *Test CrowdStrike next-gen AV for yourself. Start your free trial of Falcon Prevent™ today.*