


# A better zip bomb

David Fifield  
[david@bamsoftware.com](mailto:david@bamsoftware.com)

2019-07-02 updated 2019-07-03, 2019-07-05, 2019-07-06, 2019-07-08, 2019-07-18, 2019-07-20, 2019-07-22, 2019-07-24, 2019-08-05, 2019-08-19, 2019-08-22, 2019-10-14, 2019-10-18, 2019-10-30, [2019-11-28](#), [2020-07-28](#), 2021-01-21, 2021-02-02, [2021-05-03](#), 2021-07-29

**Summary** This article shows how to construct a *non-recursive* [zip bomb](#) that achieves a high compression ratio by overlapping files inside the zip container. "Non-recursive" means that it does not rely on a decompressor's recursively unpacking zip files nested within zip files: it expands fully after a single round of decompression. The output size increases quadratically in the input size, reaching a compression ratio of over 28 million (10 MB → 281 TB) at the limits of the zip format. Even greater expansion is possible using 64-bit extensions. The construction uses only the most common compression algorithm, DEFLATE, and is compatible with most zip parsers.

	<a href="#">zbsm.zip</a>	42 kB	→	5.5 GB
	<a href="#">zblg.zip</a>	10 MB	→	281 TB
	<a href="#">zbxl.zip</a>	46 MB	→	4.5 PB (Zip64, less compatible)

Source code:  
`git clone https://www.bamsoftware.com/git/zipbomb.git`  
[zipbomb-20210121.zip](#)  
Data and source for figures:  
`git clone https://www.bamsoftware.com/git/zipbomb-paper.git`

[Presentation video](#)  
  
[Русский перевод](#) от [@mlrko](#).  
  
[中文翻译](#): 北岸冷若冰霜.

	non-recursive			recursive	
	zipped size	unzipped size	ratio	unzipped size	ratio
<a href="#">Cox quine</a>	440	440	1.0	∞	∞
<a href="#">Ellingsen quine</a>	28 809	42 569	1.5	∞	∞
<a href="#">42.zip</a>	*42 374	558 432	13.2	4 507 981 343 026 016	106 billion
this technique	42 374	5 461 307 620	129 thousand	5 461 307 620	129 thousand
this technique	9 893 525	281 395 456 244 934	28 million	281 395 456 244 934	28 million
this technique (Zip64)	45 876 952	4 507 981 427 706 459	98 million	4 507 981 427 706 459	98 million

\* There are two versions of 42.zip, an [older version](#) of 42 374 bytes, and a [newer version](#) of 42 838 bytes. The difference is that the newer version requires a password before unzipping. We compare only against the older version. Here is a copy if you need it: [42.zip](#).

I would like to know/credit the maker of 42.zip but haven't been able to find a source—[let me know](#) if you have any info.

Compression bombs that use the zip format must cope with the fact that DEFLATE, the compression algorithm most commonly supported by zip parsers, [cannot achieve](#) a compression ratio greater than 1032. For this reason, zip bombs typically rely on recursive decompression, nesting zip files within zip files to get an extra factor of 1032 with each layer. But the trick only works on implementations that unzip recursively, and most do not. The best-known zip bomb, [42.zip](#), expands to a formidable 4.5 PB if all six of its layers are recursively unzipped, but a trifling 0.6 MB at the top layer. Zip quines, like those of [Ellingsen](#) and [Cox](#), which contain a copy of themselves and thus expand infinitely if recursively unzipped, are likewise perfectly safe to unzip once.

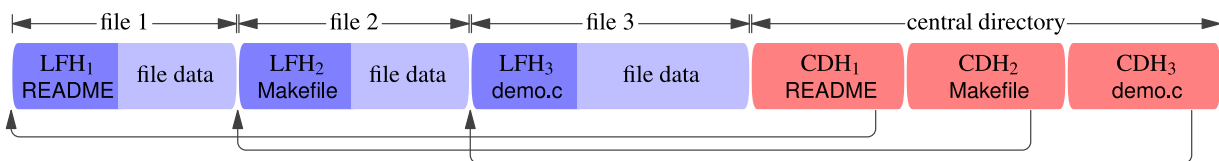
This article shows how to construct a non-recursive zip bomb whose compression ratio surpasses the DEFLATE limit of 1032. It works by overlapping files inside the zip container, in order to reference a "kernel" of highly compressed data in multiple files, without making multiple copies of it. The zip bomb's output size grows quadratically in the input size; i.e.,

the compression ratio gets better as the bomb gets bigger. The construction depends on features of both zip and DEFLATE—it is not directly portable to other file formats or compression algorithms. It is compatible with most zip parsers, the exceptions being "streaming" parsers that parse in one pass without first consulting the zip file's central directory. We try to balance two conflicting goals:

- Maximize the compression ratio. We define the compression ratio as the the sum of the sizes of all the files contained the in the zip file, divided by the size of the zip file itself. It does not count filenames or other filesystem metadata, only contents.
- Be compatible. Zip is a tricky format and parsers differ, especially around edge cases and optional features. Avoid taking advantage of tricks that only work with certain parsers. We will remark on certain ways to increase the efficiency of the zip bomb that come with some loss of compatibility.

## Structure of a zip file

A zip file consists of a *central directory* which references *files*.



The central directory is at the end of the zip file. It is a list of *central directory headers*. Each central directory header contains metadata for a single file, like its filename and CRC-32 checksum, and a backwards pointer to a local file header. A central directory header is 46 bytes long, plus the length of the filename.

A file consists of a *local file header* followed by compressed *file data*. The local file header is 30 bytes long, plus the length of the filename. It contains a redundant copy of the metadata from the central directory header, and the compressed and uncompressed sizes of the file data that follows. Zip is a container format, not a compression algorithm. Each file's data is compressed using an algorithm specified in the metadata—usually [DEFLATE](#).

The many redundancies and ambiguities in the zip format allow for all kinds of mischief. The zip bomb is just scratching the surface. Links for further reading:

[Ten thousand security pitfalls: The ZIP file format](#), talk by Gynael Coldwind

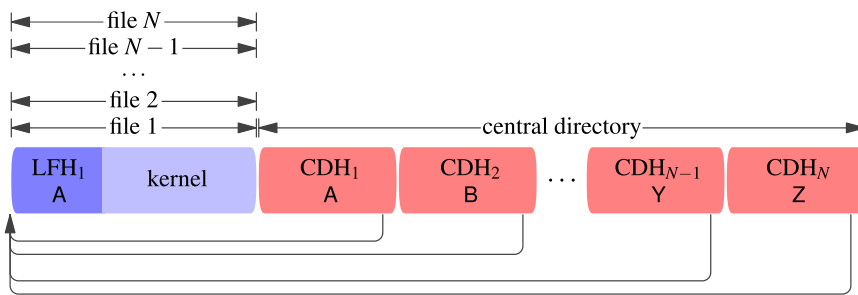
[Zip - How not to design a file format](#), by Gregg Tavares

[Ambiguous zip parsing allows hiding add-on files from linter and reviewers](#), a vulnerability I found in addons.mozilla.org

This description of the zip format omits many details that are not needed for understanding the zip bomb. For full information, refer to [section 4.3 of APPNOTE.TXT](#) or [The structure of a PKZip file](#) by Florian Buchholz, or see the [source code](#).

## The first insight: overlapping files

By compressing a long string of repeated bytes, we can produce a *kernel* of highly compressed data. By itself, the kernel's compression ratio cannot exceed the DEFLATE limit of 1032, so we want a way to reuse the kernel in many files, without making a separate copy of it in each file. We can do it by overlapping files: making many central directory headers point to a single file, whose data is the kernel.



Let's look at an example to see how this construction affects the compression ratio. Suppose the kernel is 1000 bytes and decompresses to 1 MB. Then the first MB of output "costs" 1078 bytes of input:

- 31 bytes for a local file header (including a 1-byte filename)
- 47 bytes for a central directory header (including a 1-byte filename)
- 1000 bytes for the kernel itself

But every 1 MB of output after the first costs only 47 bytes—we don't need another local file header or another copy of the kernel, only an additional central directory header. So while the first reference of the kernel has a compression ratio of  $1\,000\,000 / 1078 \approx 928$ , each additional reference pulls the ratio closer to  $1\,000\,000 / 47 \approx 21\,277$ . A bigger kernel raises the ceiling.

The problem with this idea is a lack of compatibility. Because many central directory headers point to a single local file header, the metadata—specifically the filename—cannot match for every file. Some parsers [balk at that](#). [Info-ZIP UnZip](#) (the standard Unix unzip program) extracts the files, but with warnings:

```
$ unzip overlap.zip
  inflating: A
B: mismatching "local" filename (A),
   continuing with "central" filename version
  inflating: B
...
```

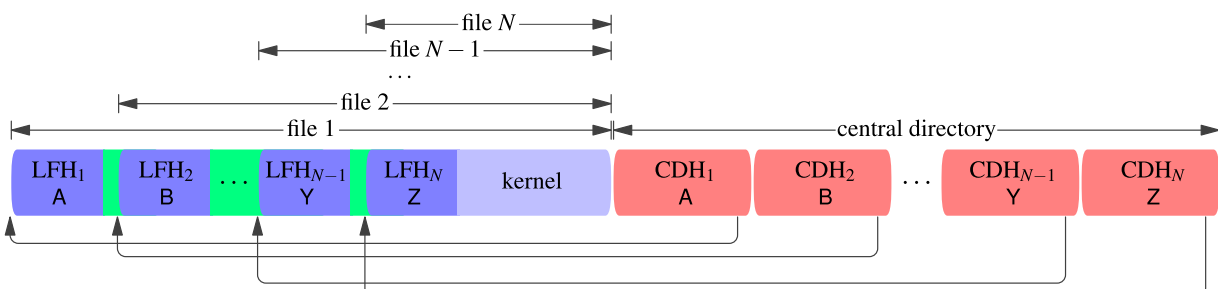
And the Python [zipfile](#) module [throws an exception](#):

```
$ python3 -m zipfile -e overlap.zip .
Traceback (most recent call last):
...
__main__.BadZipFile: File name in directory 'B' and header b'A' differ.
```

Next we will see how to modify the construction for consistency of filenames, while still retaining most of the advantage of overlapping files.

## The second insight: quoting local file headers

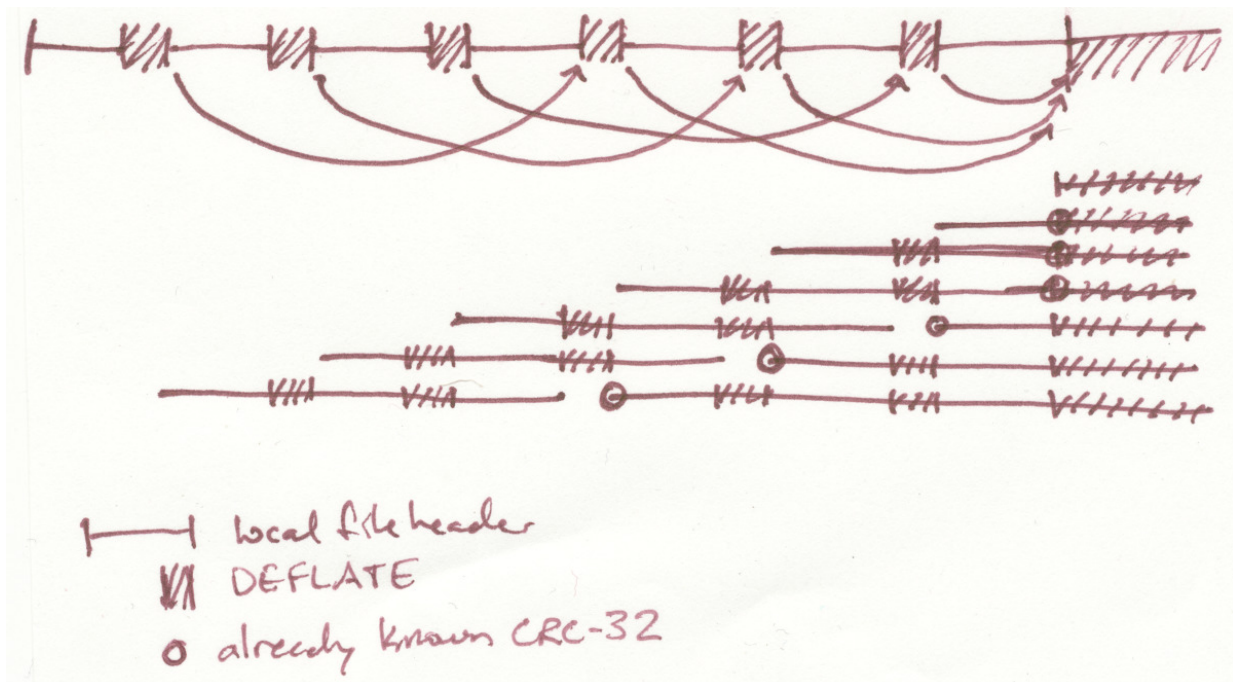
We need to separate the local file headers for each file, while still reusing a single kernel. Simply concatenating all the local file headers does not work, because the zip parser will find a local file header where it expects to find the beginning of a DEFLATE stream. But the idea will work, with a minor modification. We'll use a feature of DEFLATE, non-compressed blocks, to "quote" local file headers so that they appear to be part of the same DEFLATE stream that terminates in the kernel. Every local file header (except the first) will be interpreted in two ways: as code (part of the structure of the zip file) and as data (part of the contents of a file).



A DEFLATE stream is a sequence of [blocks](#), where each block may be compressed or non-compressed. Compressed blocks are what we usually think of; for example the kernel is one big compressed block. But there are also non-compressed blocks, which start with a [5-byte header](#) with a length field that means simply, "output the next  $n$  bytes verbatim." Decompressing a non-compressed block means only stripping the 5-byte header. Compressed and non-compressed blocks may be intermixed freely in a DEFLATE stream. The output is the concatenation of decompressing all the blocks in order. The "non-compressed" notion only has meaning at the DEFLATE layer; the file data still counts as "compressed" at the zip layer, no matter what kind of blocks are used.

It is easiest to understand this quoted-overlap construction from the inside out, beginning with the last file and working backwards to the first. Start by inserting the kernel, which will form the end of file data for every file. Prepend a local file header  $\text{LFH}_N$  and add a central directory header  $\text{CDH}_N$  that points to it. Set the "compressed size" metadata field in the  $\text{LFH}_N$  and  $\text{CDH}_N$  to the compressed size of the kernel. Now prepend a 5-byte non-compressed block header (colored green in the diagram) whose length field is equal to the size of  $\text{LFH}_N$ . Prepend a second local file header  $\text{LFH}_{N-1}$  and add a central directory header  $\text{CDH}_{N-1}$  that points to it. Set the "compressed size" metadata field in both of the new headers to the compressed size of the kernel *plus* the size of the non-compressed block header (5 bytes) *plus* the size of  $\text{LFH}_N$ .

2019-08-22: There's an additional minor optimization possible that I didn't originally think of. Instead of only quoting the immediately following local file header, quote as many local file headers as possible—including their own quoting blocks—up to the limit of 65535 bytes per non-compressed block. The advantage is that the quoting blocks between local file headers now additionally become part of the output file, gaining 5 bytes of output for each one we manage to include. It's a small optimization, gaining only 154 380 bytes in `zbsm.zip`, or 0.003%. (Far less than [extra-field quoting](#) gains.) The `--giant-steps` option in the [source code](#) activates this feature.



The giant-steps feature only pays when you are not constrained by maximum output file size. In `zblg.zip`, we actually want to slow file growth as much as possible so that the smallest file, containing the kernel, can be as large as possible. Using giant steps in `zblg.zip` actually decreases the compression ratio.

I credit [Kevin Farrow](#) for sparking the idea for this enhancement during a [dc303 talk](#). Carlos Javier González Cortés (Lethani) also hit on the idea in [his article \(Español\)](#) on overlapped zip bombs.

At this point the zip file contains two files, named "Y" and "Z". Let's walk through what a zip parser would see while parsing it. Suppose the compressed size of the kernel is 1000 bytes and the size of  $\text{LFH}_N$  is 31 bytes. We start at  $\text{CDH}_{N-1}$  and follow the pointer to  $\text{LFH}_{N-1}$ . The first file's filename is "Y" and the compressed size of its file data is 1036 bytes. Interpreting the next 1036 bytes as a DEFLATE stream, we first encounter the 5-byte header of a non-compressed block that says to copy the next 31 bytes. We write the next 31 bytes, which are  $\text{LFH}_N$ , which we decompress and append to file "Y". Moving on in the DEFLATE stream, we find a compressed block (the kernel), which we decompress to file "Y". Now we have reached the end of the compressed data and are done with file "Y". Proceeding to the next file, we follow the pointer from  $\text{CDH}_N$  to  $\text{LFH}_N$  and find a file named "Z" whose compressed size is 1000 bytes. Interpreting those 1000 bytes as a DEFLATE stream, we immediately encounter a compressed block (the kernel again) and decompress it to the file "Z". Now we have reached the end of the final file and are done. The output

file "Z" contains the decompressed kernel; the output file "Y" is the same, but additionally prefixed by the 31 bytes of  $\text{LFH}_N$ .

We complete the construction by repeating the quoting procedure until the zip file contains the desired number of files. Each new file adds a central directory header, a local file header, and a non-compressed block to quote the immediately succeeding local file header. Compressed file data is generally a chain of DEFLATE non-compressed blocks (the quoted local file headers) followed by the compressed kernel. Each byte in the kernel contributes about  $1032 N$  to the output size, because each byte is part of all  $N$  files. The output files are not all the same size: those that appear earlier in the zip file are larger than those that appear later, because they contain more quoted local file headers. The contents of the output files are not particularly meaningful, but no one said they had to make sense.

This quoted-overlap construction has better compatibility than the full-overlap construction of the previous section, but the compatibility comes at the expense of the compression ratio. There, each added file cost only a central directory header; here, it costs a central directory header, a local file header, and another 5 bytes for the quoting header.

## Optimization

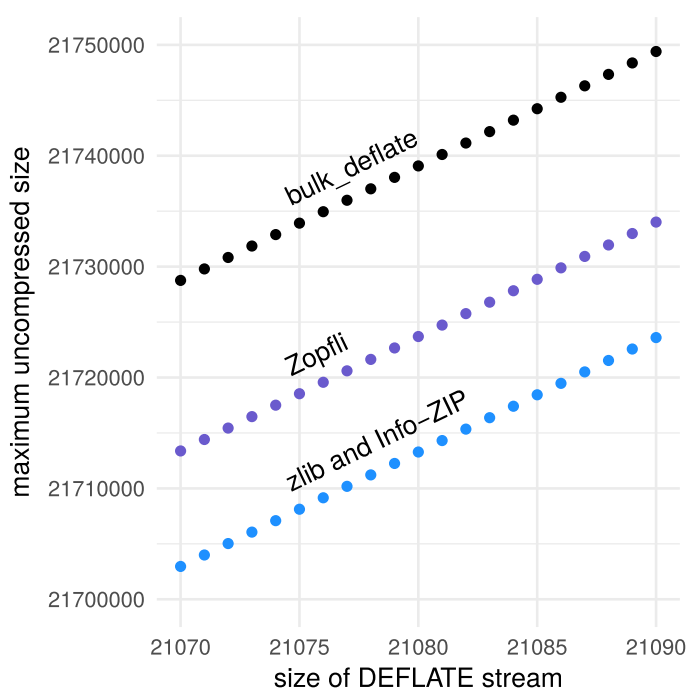
Now that we have the basic zip bomb construction, we will try to make it as efficient as possible. We want to answer two questions:

- For a given zip file size, what is the maximum compression ratio?
- What is the maximum compression ratio, given the limits of the zip format?

## Kernel compression

It pays to compress the kernel as densely as possible, because every decompressed byte gets magnified by a factor of  $N$ . To that end, we use a custom DEFLATE compressor called `bulk_deflate`, specialized for compressing a string of repeated bytes.

All decent DEFLATE compressors will approach a compression ratio of 1032 when given an infinite stream of repeating bytes, but we care more about specific finite sizes than asymptotics. `bulk_deflate` compresses more data into the same space than the general-purpose compressors: about 26 kB more than `zlib` and `Info-ZIP`, and about 15 kB more than [Zopfli](https://zopfli.com/), a compressor that trades speed for density.



The price of `bulk_deflate`'s high compression ratio is a lack of generality. `bulk_deflate` can only compress strings of a single repeated byte, and only those of specific lengths, namely  $517 + 258k$  for integer  $k \geq 0$ . Besides compressing

densely, `bulk_deflate` is fast, doing essentially constant work regardless of the input size, aside from the work of actually writing out the compressed string.

## Filenames

Every byte spent on a filename is 2 bytes not spent on the kernel. (2 because each filename appears twice, in the central directory header and the local file header.) A filename byte results in, on average, only  $(N + 1) / 4$  bytes of output, while a byte in the kernel counts for  $1032 N$ .

For our purposes, filenames are mostly dead weight. While filenames do contribute something to the output size by virtue of being part of quoted local file headers, a byte in a filename does not contribute nearly as much as a byte in the kernel. We want filenames to be as short as possible, while keeping them all distinct, and subject to compatibility considerations.

Examples: [1](#) [2](#) [3](#)

The first compatibility consideration is character encoding. The zip format specification states that filenames are to be interpreted as [CP 437](#), or [UTF-8](#) if a certain flag bit is set ([APPNOTE.TXT Appendix D](#)). But this is a major point of incompatibility across zip parsers, which may interpret filenames as being in some fixed or locale-specific encoding. So for compatibility, we must limit ourselves to characters that have the same encoding in both CP 437 and UTF-8; namely, the 95 printable characters of US-ASCII.

One thing I didn't consider is [Windows reserved filenames like "PRN" and "NUL"](#).

We are further restricted by filesystem naming limitations. Some filesystems are case-insensitive, so "a" and "A" do not count as distinct names. Common filesystems like FAT32 [prohibit certain characters](#) like '\*' and '?'.  
 As a safe but not necessarily optimal compromise, our zip bomb will use filenames consisting of characters drawn from a 36-character alphabet that does not rely on case distinctions or use special characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Filenames are generated in the obvious way, cycling each position through the possible characters and adding a position on overflow:

```
"0", "1", "2", ..., "Z",
"00", "01", "02", ..., "0Z",
...,
"Z0", "Z1", "Z2", ..., "ZZ",
"000", "001", "002", ...
```

There are 36 filenames of length 1,  $36^2$  filenames of length 2, and so on. The length of the  $n$ th filename is  $\lceil \log_{36}((n + 1) / (36 / 35)) \rceil + 1$ . Four bytes are enough to represent 1 727 604 distinct filenames.

Given that the  $N$  filenames in the zip file are generally not all of the same length, which way should we order them, shortest to longest or longest to shortest? A little reflection shows that it is better to put the longest names last, because those names are the most quoted. Ordering filenames longest last adds over 900 MB of output to [zblg.zip](#), compared to ordering them longest first. It is a minor optimization, though, as those 900 MB comprise only 0.0003% of the total output size.

## Kernel size

The quoted-overlap construction allows us to place a compressed kernel of data, and then cheaply copy it many times. For a given zip file size  $X$ , how much space should we devote to storing the kernel, and how much to making copies?

To find the optimum balance, we only have to optimize the single variable  $N$ , the number of files in the zip file. Every value of  $N$  requires a certain amount of overhead for central directory headers, local file headers, quoting block headers, and filenames. All the remaining space can be taken up by the kernel. Because  $N$  has to be an integer, and you can only



fit so many files before the kernel size drops to zero, it suffices to test every possible value of  $N$  and select the one that yields the most output.

Applying the optimization procedure to  $X = 42\,374$ , the size of 42.zip, finds a maximum at  $N = 250$ . Those 250 files require 21 195 bytes of overhead, leaving 21 179 bytes for the kernel. A kernel of that size decompresses to 21 841 249 bytes (a ratio of 1031.3). The 250 copies of the decompressed kernel, plus the little bit extra that comes from the quoted local file headers, produces an overall unzipped output of 5 461 307 620 bytes and a compression ratio of 129 thousand.



**zbsm.zip** 42 kB → 5.5 GB

```
zipbomb --mode=quoted_overlap --num-files=250 --compressed-size=21179 > zbsm.zip
```

Optimization produced an almost even split between the space allocated to the kernel and the space allocated to file headers. It is not a coincidence. Let's look at a simplified model of the quoted-overlap construction. In the simplified model, we ignore filenames, as well as the slight increase in output file size due to quoting local file headers. Analysis of the simplified model will show that the optimum split between kernel and file headers is approximately even, and that the output size grows quadratically when allocation is optimal.

Define some constants and variables:

$X$	zip file size (take as fixed)
$N$	number of files in the zip file (variable to optimize)
$CDH = 46$	size of a central directory header (without filename)
$LFH = 30$	size of a local file header (without filename)
$Q = 5$	the size of DEFLATE non-compressed block header
$C \approx 1032$	compression ratio of the kernel

Let  $H(N)$  be the amount of header overhead required by  $N$  files. Refer to [the diagram](#) to understand where this formula comes from.

$$H(N) = N \cdot (CDH + LFH) + (N - 1) \cdot Q$$

The space remaining for the kernel is  $X - H(N)$ . The total unzipped size  $S_X(N)$  is the size of  $N$  copies of the kernel, decompressed at ratio  $C$ . (In this simplified model we ignore the minor additional expansion from quoted local file headers.)

$$\begin{aligned} S_X(N) &= (X - H(N)) C N \\ &= (X - (N \cdot (CDH + LFH) + (N - 1) \cdot Q)) C N \\ &= -(CDH + LFH + Q) C N^2 + (X + Q) C N \end{aligned}$$

$S_X(N)$  is a polynomial in  $N$ , so its maximum must be at a place where the derivative  $S'_X(N)$  is zero. Taking the derivative and finding the zero gives us  $N_{OPT}$ , the optimal number of files.

$$\begin{aligned} S'_X(N_{OPT}) &= -2 (CDH + LFH + Q) C N_{OPT} + (X + Q) C \\ 0 &= -2 (CDH + LFH + Q) C N_{OPT} + (X + Q) C \\ N_{OPT} &= (X + Q) / (CDH + LFH + Q) / 2 \end{aligned}$$

$H(N_{OPT})$  gives the optimal amount of space to allocate for file headers. It is independent of  $CDH$ ,  $LFH$ , and  $C$ , and is close to  $X/2$ .

$$\begin{aligned} H(N_{OPT}) &= N_{OPT} \cdot (CDH + LFH) + (N_{OPT} - 1) \cdot Q \\ &= (X - Q) / 2 \end{aligned}$$

$S_X(N_{OPT})$  is the total unzipped size when the allocation is optimal. From this we see that the output size grows quadratically in the input size.

$$S_X(N_{OPT}) = (X + Q)^2 C / (CDH + LFH + Q) / 4$$

It's a little more complicated, because the precise limits depend on the implementation. Python zipfile [ignores](#) the number of files. Go archive/zip [allows](#) larger file counts, as long as they are equal in the lower 16 bits. But for broad compatibility, we have to stick to the limits as stated.

As we make the zip file larger, eventually we run into the limits of the zip format. A zip file can contain at most  $2^{16} - 1$  files, and each file can have an uncompressed size of at most  $2^{32} - 1$  bytes. Worse than that, [some implementations](#) take the maximum possible values as an indicator of the presence of [64-bit extensions](#), so our limits are actually  $2^{16} - 2$  and  $2^{32} - 2$ . It happens that the first limit we hit is the one on uncompressed file size. At a zip file size of 8 319 377 bytes, naive optimization would give us a file count of 47 837 and a largest file of  $2^{32} + 311$  bytes.

Accepting that we cannot increase  $N$  nor the size of the kernel without bound, we would like find the maximum compression ratio achievable while remaining within the limits of the zip format. The way to proceed is to make the kernel as large as possible, and have the maximum number of files. Even though we can no longer maintain the roughly even split between kernel and file headers, each added file *does* increase the compression ratio—just not as fast as it would if we were able to keep growing the kernel, too. In fact, as we add files we will need to *decrease* the size of the kernel to make room for the maximum file size that gets slightly larger with each added file.

The plan results in a zip file that contains  $2^{16} - 2$  files and a kernel that decompresses to  $2^{32} - 2$  178 825 bytes. Files get longer towards the beginning of the zip file—the first and largest file decompresses to  $2^{32} - 56$  bytes. That is as close as we can get using the coarse output sizes of bulk\_deflate—encoding the final 54 bytes would cost more bytes than they are worth. (The zip file as a whole has a compression ratio of 28 million, and the final 54 bytes would gain at most  $54 \cdot 1032 \cdot (2^{16} - 2) \approx 36.5$  million bytes, so it only helps if the 54 bytes can be encoded in 1 byte—I could not do it in less than 2.) The output size of this zip bomb, 281 395 456 244 934 bytes, is 99.97% of the theoretical maximum  $(2^{32} - 1) \cdot (2^{16} - 1)$ . Any major improvements to the compression ratio can only come from reducing the input size, not increasing the output size.



**zblg.zip** 10 MB → 281 TB

```
zipbomb --mode=quoted_overlap --num-files=65534 --max-uncompressed-size=4292788525 > zblg.zip
```

## Efficient CRC-32 computation

Among the metadata in the central directory header and local file header is a [CRC-32](#) checksum of the uncompressed file data. This poses a problem, because directly calculating the CRC-32 of each file requires doing work proportional to the total *unzipped* size, which is large by design. (It's a zip bomb, after all.) We would prefer to do work that in the worst case is proportional to the *zipped* size. Two factors work in our advantage: all files share a common suffix (the kernel), and the uncompressed kernel is a string of repeated bytes. We will represent CRC-32 as a matrix product—this will allow us not only to compute the checksum of the kernel quickly, but also to reuse computation across files. The technique described in this section is a slight extension of the [crc32\\_combine](#) function in zlib, which Mark Adler explains [here](#).

You can model CRC-32 as a state machine that updates a 32-bit state register for each incoming bit. The basic update operations for a 0 bit and a 1 bit are:

```
uint32 crc32_update_0(uint32 state) {
    // Shift out the least significant bit.
    bit b = state & 1;
    state = state >> 1;
    // If the shifted-out bit was 1, XOR with the CRC-32 constant.
    if (b == 1)
        state = state ^ 0xedb88320;
    return state;
}

uint32 crc32_update_1(uint32 state) {
    // Do as for a 0 bit, then XOR with the CRC-32 constant.
    return crc32_update_0(state) ^ 0xedb88320;
}
```

If you think of the state register as a 32-element binary vector, and use XOR for addition and AND for multiplication, then `crc32_update_0` is a [linear transformation](#); i.e., it can be represented as multiplication by a  $32 \times 32$  binary [transformation matrix](#). To see why, observe that multiplying a matrix by a vector is just summing the columns of the matrix, after multiplying each column by the corresponding element of the vector. The shift operation `state >> 1` is just taking each bit  $i$  of the state vector and multiplying it by a vector that is 0 everywhere except at bit  $i - 1$  (numbering the



Furthermore, `crc32_update_1` is just `crc32_update_0` plus (XOR) a constant. That makes `crc32_update_1` an [affine transformation](#): a matrix multiplication followed by a translation (i.e., vector addition). We can represent both the matrix multiplication and the translation in a single step if we enlarge the dimensions of the transformation matrix to  $33 \times 33$  and append an extra element to the state vector that is always 1. (This representation is called [homogeneous coordinates](#).)

$$M_0$$
$$M_1$$

operations `crc32_update_0` and `crc32_update_1` can be represented by a  $33 \times 33$  transformation matrix. The matrices  $M_0$  and  $M_1$  are shown. The benefit of a matrix representation is that matrices compose. Suppose we want to represent the change effected by processing the ASCII character 'a', whose binary representation is  $01100001_2$ . We can represent cumulative CRC-32 state change of those 8 bits in a single transformation matrix:

We can represent the state change of a string of repeated 'a's by multiplying many copies of  $M_a$  together—matrix exponentiation. We can do matrix exponentiation quickly using a [square-and-multiply](#) algorithm, which allows us to compute  $M^n$  in only about  $\log_2 n$  steps. For example, the matrix representing the state change of a string of 9 'a's is

square-and-multiply algorithm is useful for computing  $M_{\text{kernel}}$ , the matrix for the uncompressed kernel, because the 1 is a string of repeated bytes. To produce a CRC-32 checksum value from a matrix, multiply the matrix by the zero vector. (The zero vector in homogeneous coordinates, that is: 32 0's followed by a 1. Here we omit the minor for brevity.) To compute the checksum for every file, we work backwards. For file  $N$ , multiply  $M$  by initializing  $M := M_{\text{kernel}}$ . The checksum of the kernel is also the checksum of the final file, file  $N$ , so multiply  $M$  by the zero vector and store the resulting checksum in  $\text{CDH}_N$  and  $\text{LFH}_N$ . The file data of file  $N - 1$  is the same as the file data of file  $N$ , but with an added prefix of  $\text{LFH}_N$ . So compute  $M_{\text{LFH}_N}$ , the state change matrix for  $\text{LFH}_N$ , and update  $M := M_{\text{LFH}_N} M$ . Now  $M$  represents the cumulative state change from processing  $\text{LFH}_N$  followed by the kernel. Compute the checksum for file  $N - 1$  by again multiplying  $M$  by the zero vector. Continue the procedure, accumulating state change matrices into  $M$ , until all the files have been processed.

## Extension: Zip64

[Earlier](#) we hit a wall on expansion due to limits of the zip format—it was impossible to produce more than about 281 TB of output, no matter how cleverly packed the zip file. It is possible to surpass those limits using [Zip64](#), an extension to the zip format that increases the size of certain header fields to 64 bits. Support for Zip64 is [by no means universal](#), but it is one of the more commonly implemented extensions. As regards the compression ratio, the effect of Zip64 is to increase the size of a central directory header from 46 bytes to 58 bytes, and the size of a local directory header from 30 bytes to 50 bytes. Referring to [the formula](#) for optimal expansion in the simplified model, we see that a zip bomb in Zip64 format still grows quadratically, but more slowly because of the larger denominator—this is visible in [the figure below](#) in the Zip64 line's slightly lower vertical placement. In exchange for the loss of compatibility and slower growth, we get the removal of all practical file size limits.

Suppose we want a zip bomb that expands to 4.5 PB, the same size that 42.zip recursively expands to. How big must the zip file be? Using binary search, we find that the smallest zip file whose unzipped size exceeds the unzipped size of 42.zip has a zipped size of 46 MB.



**zbx1.zip** 46 MB → 4.5 PB (Zip64, less compatible)

```
zipbomb --mode=quoted_overlap --num-files=190023 --compressed-size=22982788 --zip64 > zbx1.zip
```

4.5 PB is roughly the size of the data captured by the Event Horizon Telescope to make the [first image of a black hole](#), stacks and stacks of hard drives.

With Zip64, it's no longer practically interesting to consider the maximum compression ratio, because we can just keep increasing the zip file size, and the compression ratio along with it, until even the compressed zip file is prohibitively large. An interesting threshold, though, is  $2^{64}$  bytes (18 EB or 16 EiB)—that much data [will not fit on most filesystems](#). Binary search finds the smallest zip bomb that produces at least that much output: it contains 12 million files and has a compressed kernel of 1.5 GB. The total size of the zip file is 2.9 GB and it unzips to  $2^{64} + 11\,727\,895\,877$  bytes, having a compression ratio of over 6.2 billion. I didn't make this one downloadable, but you can generate it yourself using the [source code](#). It contains files so large that it uncovers [a bug](#) in Info-ZIP UnZip 6.0.

```
zipbomb --mode=quoted_overlap --num-files=12056313 --compressed-size=1482284040 --zip64 > zbx1.zip
```

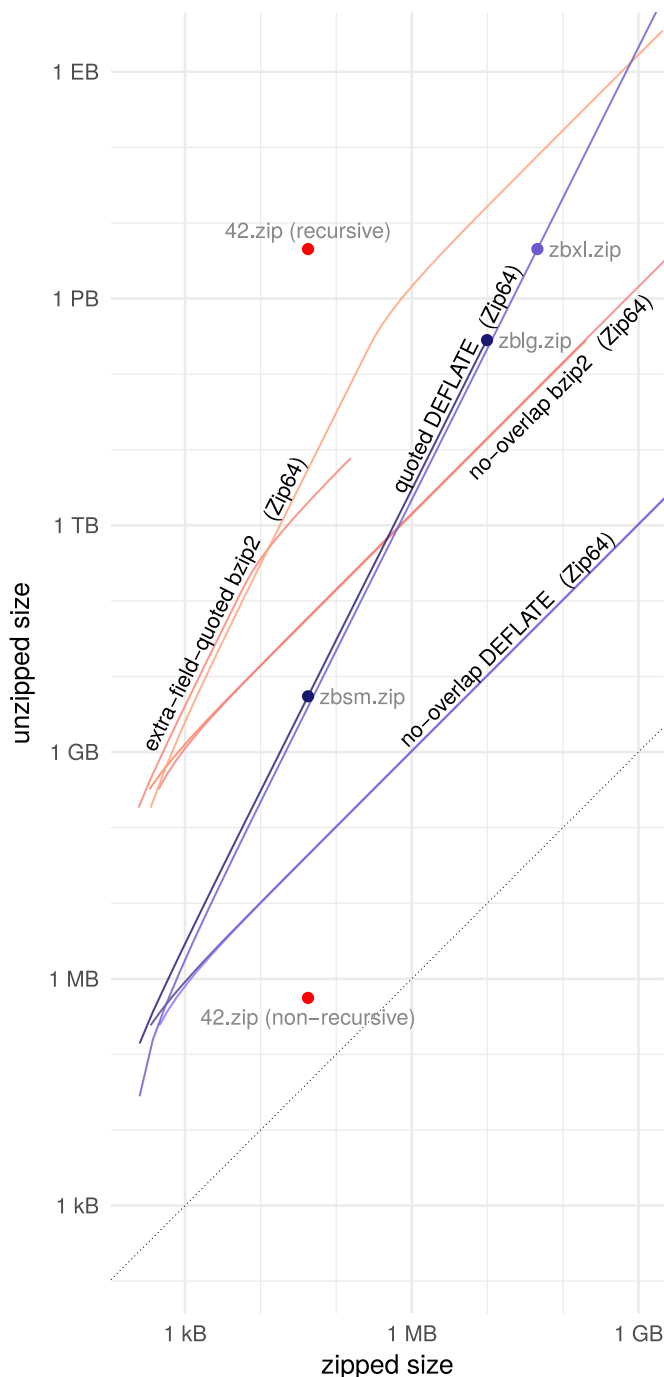
## Extension: bzip2

bzip2 starts with a run-length encoding step that reduces the length of a string of repeated bytes by a factor of 51. Then the data is separated into 900 kB blocks and each block compressed individually. Empirically, one block after run-length encoding can compress down to 32 bytes.  $900\,000 \times 51 / 32 = 1\,434\,375$ .

DEFLATE is the most common compression algorithm used in the zip format, but it is only one of many options. Probably the second most common algorithm is [bzip2](#), while not as compatible as DEFLATE, is probably the second most commonly supported compression algorithm. Empirically, bzip2 has a maximum compression ratio of about 1.4 million, which allows for denser packing of the kernel. Ignoring the loss of compatibility, does bzip2 enable a more efficient zip bomb?

Yes—but only for small files. The problem is that bzip2 does not have anything like the [non-compressed blocks](#) of DEFLATE that we used to [quote local file headers](#). So it is not possible to overlap files and reuse the kernel—each file must have its own copy, and therefore the overall compression ratio is no better than the ratio of any single file. In [the figure](#) we see that no-overlap bzip2 outperforms quoted DEFLATE only for files under about a megabyte.

There is still hope for using bzip2—an alternative means of local file header quoting discussed in [the next section](#). Additionally, if you happen to know that a certain zip parser supports bzip2 *and* tolerates mismatched filenames, then you can use the [full-overlap construction](#), which has no need for quoting.



Zipped size versus unzipped size for various zip bomb constructions. Note the log-log scales. Each construction is shown with and without Zip64. The no-overlap constructions have a linear rate of growth, which is visible in the 1:1 slope of the lines. The vertical offset of the bzip2 lines shows that the compression ratio of bzip2 is about a thousand times greater than that of DEFLATE. The quoted-DEFLATE constructions have a quadratic rate of growth, as evidenced by the 2:1 slope of the lines. The Zip64 variant is slightly less efficient, but permits output in excess of 281 TB. The lines for extra-field-quoted bzip2 transition from quadratic to linear upon reaching either the maximum file size ( $2^{32} - 2$  bytes), or the maximum number of files allowed by extra-field quoting.

## Extension: extra-field quoting

So far we have used a feature of DEFLATE to quote local file headers, and we have just seen that the same trick does not work with bzip2. There is an alternative means of quoting, somewhat more limited, that only uses features of the zip format and does not depend on the compression algorithm.

At the end of the local file header structure there is a variable-length *extra field* whose purpose is to store information that doesn't fit into the ordinary fields of the header ([APPNOTE.TXT section 4.3.7](#)). The extra information may include,

for example, a high-resolution timestamp or a Unix uid/gid; Zip64 works by using the extra field. The extra field is a length-value structure: if we increase the length field without adding to the value, then it will grow to include whatever comes after it in the zip file—namely the next local file header. Each local file header "quotes" the local file headers that follow it by enclosing them within its own extra field. The benefits of extra-field quoting over DEFLATE quoting are threefold:

1. Extra-field quoting requires only 4 bytes of overhead, not 5, leaving more room for the kernel.
2. Extra-field quoting does not increase the size of files, which leaves more headroom for a bigger kernel when operating at the limits of the zip format.
3. Extra-field quoting provides a way to combine quoting with bzip2.

The [zipalign](#) tool from Android aligns files to 4-byte boundaries. It works by [padding the extra field with 0x00 bytes](#). Thus it could be considered to use header ID 0x0000, which is "reserved for use by PKWARE." But because it may add 0, 1, 2, or 3 bytes of padding, and an extra field header is 4 bytes, the extra fields it produces may be invalid anyway.

Despite these benefits, extra-field quoting is less flexible than DEFLATE quoting. It does not chain: each local file header must enclose not only the immediately next header but *all* headers which follow. The extra fields increase in length as they get closer to the beginning of the zip file. Because the extra field has a maximum length of  $2^{16} - 1$  bytes, it can only contain up to 1808 local file headers, or 1170 with Zip64, assuming that filenames are [allocated as described](#). (With DEFLATE, you can use extra-field quoting for the earliest local file headers, then switch to DEFLATE quoting for the remainder.) Another problem is that, in order to conform to the internal data structure of the extra field, you must select a 16-bit *header ID* ([APPNOTE.TXT section 4.5.2](#)), to precede the quoted data. We want a header ID that will make parsers ignore the quoted data, not try to interpret it as meaningful metadata. Zip parsers are supposed to ignore unknown header IDs, so we could choose one at random, but there is the risk that the ID may be allocated in the future, breaking compatibility.

[The figure](#) illustrates the possibility of combining extra-field quoting with bzip2, with and without Zip64. Both "extra-field-quoted bzip2" lines have a knee at which the growth transitions from quadratic to linear. In the non-Zip64 case, the knee occurs at the maximum uncompressed file size ( $2^{32} - 2$  bytes); after this point, one can only increase the number of files, not their size. The line stops completely when the number of files reaches 1809, and we run out of room in the extra field. In the Zip64 case, the knee occurs at 1171 files, after which the size of files can be increased, but not their number. Extra-field quoting may also be used with DEFLATE, but the improvement is so slight that it has been omitted from the figure. It increases the compression ratio of zbsm.zip by 1.2%; zblg.zip by 0.019%; and zbx1.zip by 0.0025%.

## Discussion

In related work, [Plötz et al.](#) used overlapping files to create a near-self-replicating zip file. Gynael Coldwind has [previously suggested](#) (slide 47) overlapping files. [Pellegrino et al.](#) found systems vulnerable to compression bombs and other resource exhaustion attacks and listed common pitfalls in specification, implementation, and configuration.

We have designed the quoted-overlap zip bomb construction for compatibility, taking into consideration a number of implementation differences, some of which are shown in [the table below](#). The resulting construction is compatible with zip parsers that work in the usual back-to-front way, first consulting the central directory and using it as an index of files. Among these is the example zip parser included in [Nail](#), which is automatically generated from a formal grammar. The construction is not compatible, however, with "streaming" parsers, those that parse the zip file from beginning to end in one pass without first reading the central directory. By their nature, streaming parsers do not permit any kind of file overlapping. The most likely outcome is that they will extract only the first file. They may even raise an error besides, as is the case with [sunzip](#), which parses the central directory at the end and checks it for consistency with the local file headers it has already seen.

If you need the extracted files to start with a certain prefix (so that they will be identified as a certain file type, for example), you can insert a data-carrying DEFLATE block just before the block that quotes the next header. Not every file has to participate in the bomb construction: you can include ordinary files alongside the bomb files if you need the zip file to conform to some higher-level format. (The [source code](#) has a `--template` option to facilitate this use case.) Many file formats use zip as a container; examples are Java JAR, Android APK, and LibreOffice documents.

[PDF](#) is in many ways similar to zip. It has a cross-reference table at the end of the file that points to objects earlier in the file, and it supports DEFLATE compression of objects through the FlateDecode filter. Didier Stevens [writes](#) about having contained a 1 GB stream inside a 2.6 kB PDF file by stacking FlateDecode filters. If a PDF parser limits the amount of stacking, then it is probably possible to use the DEFLATE quoting idea to overlap PDF objects.

The central directory is located indirectly using another structure called the *end of central directory* (EOCD). The EOCD ends in a variable-length comment field and finding it requires scanning for a magic number. libziparchive [calls](#) the process the "traditional EOCD snipe hunt" :)

Detecting the specific class of zip bomb we have developed in this article is easy: look for overlapping files. Mark Adler has written [a patch](#) for Info-ZIP UnZip that does just that. In general, though, rejecting overlapping files does not by itself make it safe to handle untrusted zip files. There are zip bombs that do not rely on overlapping files, and there are malicious zip files that are not bombs. Furthermore, any such detection logic must be implemented inside the parser itself, not as a separate prefilter. One of the details omitted from [the description of the zip format](#) is that there is no single well-defined algorithm for locating the central directory in a zip file: two parsers may find two different central directories and therefore [may not even agree on what files a zip file contains](#) (slides 67–80). Predicting the total uncompressed size by summing the sizes of all files does not work, in general, because the sizes stored in metadata [may not match](#) (§4.2.2) the actual uncompressed sizes. (See the "permits too-short file size" row in [the compatibility table](#).) Robust protection against zip bombs involves sandboxing the parser to limit its use of time, memory, and disk space—just as if you were processing image files, or any other complex file format prone to parser bugs.

	<a href="#">Info-ZIP UnZip 6.0</a>	<a href="#">Python 3.7 zipfile</a>	<a href="#">Go 1.12 archive/zip</a>	<a href="#">yauzl 2.10.0 (Node.js)</a>	<a href="#">Nail examples/zip</a>	<a href="#">Android 9.0.0 r1 libziparchive</a>	<a href="#">sunzip 0.4 (streaming)</a>
DEFLATE	✓	✓	✓	✓	✓	✓	✓
Zip64	✓	✓	✓	✓	✗	✗	✓
bzip2	✓	✓	✗	✗	✗	✗	✓
permits mismatched filenames	warns	✗	✓	✓	✓	✗	✓
permits incorrect CRC-32	warns	✗	if zero	✓	✗	✓	✗
permits too-short file size	✓	✗	✗	✗	✗	✗	✗
permits file size of $2^{32} - 1$	✓	✓	✓	✗	✓	✓	✓
permits file count of $2^{16} - 1$	✓	✓	✓	✗	✓	✓	✓
unzips <a href="#">overlap.zip</a>	warns	✗	✓	✓	✓	✗	✗
unzips <a href="#">zbsm.zip</a> and <a href="#">zblg.zip</a>	✓	✓	✓	✓	✓	✓	✗
unzips <a href="#">zbxl.zip</a>	✓	✓	✓	✓	✗	✗	✗

Compatibility of selected zip parsers with various zip features, edge cases, and zip bomb constructions. The background colors indicate a scale from less restrictive to more restrictive. For best compatibility, use DEFLATE compression without Zip64, match names in central directory headers and local file headers, compute correct CRCs, and avoid the maximum values of 32-bit and 16-bit fields.

## Credits

I thank [Mark Adler](#), [Blake Burkhart](#), [Gynvael Coldwind](#), [Russ Cox](#), [Brandon Enright](#), [Joran Dirk Greef](#), [Marek Majkowski](#), [Josh Wolfe](#), and the [USENIX WOOT 2019](#) reviewers for comments on this article or a draft. Caolán McNamara evaluated the security impact of the zip bombs in LibreOffice. [@m1rko](#) wrote a [Russian translation](#). [北岸冷若冰霜](#) wrote a [Chinese translation](#). Daniel Ketterer reported that the `--template` option was broken after the addition of [--giant-steps](#).

A version of this article appeared at the [USENIX WOOT 2019](#) workshop. The workshop talk [video](#), [slides](#), and [transcript](#) are available. The [source code](#) of the paper is available. The [artifacts](#) prepared for submission are [zipbomb-woot19.zip](#).

Did you find a system that chokes on one of these zip bombs? Did they help you demonstrate a vulnerability or win a bug bounty? [Let me know](#) and I'll try to mention it here.

LibreOffice 6.1.5.2

zblg.zip renamed to zblg.odt or zblg.docx will cause LibreOffice to create and delete a number of ~4 GB temporary files as it attempts to determine the file format. It does eventually finish, and it deletes the temporary files as it goes, so it's only a temporary DoS that doesn't fill up the disk. Caolán McNamara replied to my bug report.

Mozilla addons-server 2019.06.06



I tried the zip bombs against a local installation of addons-server, which is part of the software behind addons.mozilla.org. The system handles it gracefully, imposing a [time limit](#) of 110 s on extraction. The zip bomb expands as fast as the disk will let it up to the time limit, but after that point the process is killed and the unzipped files are eventually automatically cleaned up.

## UnZip 6.0

Mark Adler wrote [a patch](#) for UnZip to detect this class of zip bomb.

2019-07-05: I noticed that [CVE-2019-13232](#) was assigned for UnZip. Personally, I would dispute that UnZip's (or any zip parser's) ability to process a zip bomb of the kind discussed here necessarily represents a security vulnerability, or even a bug. It's a natural implementation and does not violate the specification in any way that I can tell. The type discussed in this article is only one type of zip bomb, and there are many ways in which zip parsing can go wrong that are not bombs. If you want to defend against resource exhaustion attacks, you should *not* try to enumerate, detect, and block every individual known attack; rather you should impose external limits on time and other resources so that the parser cannot misbehave too much, no matter what kind of attack it faces. There is nothing wrong with attempting to detect and reject certain constructions as a first-pass optimization, but you can't stop there. If you do not eventually isolate and limit operations on untrusted data, your system is likely still vulnerable. Consider an analogy with [cross-site scripting](#) in HTML: the right defense is not to try and filter out bytes that may be interpreted as code, it's to escape everything properly.

Mark Adler's patch made its way into Debian in [bug #931433](#). There were some unanticipated consequences: problems parsing certain Java JARs ([bug #931895](#)) and problems with the mutant zip format of Firefox's omni.jar file ([bug #932404](#)). SUSE decided [not to do anything](#) about CVE-2019-13232. I think both Debian's and SUSE's choices are defensible.

## ronomon/zip

Shortly after the publication of this article, Joran Dirk Greef published a [restrictive zip parser](#) (JavaScript) that prohibits irregularities such as overlapping files or unused space between files. While it may thereby reject certain valid zip files, the idea is to ensure that any downstream parsers will receive only clean, easy-to-parse files.

## antivirus engines

Overall, it seems that malware scanners have slowly begun to recognize zip bombs of this kind (or at least the specific samples available for download) as malicious. It would be interesting to see whether the detection is robust or brittle. You could reverse the order of the entries in the central directory, for example, and see whether the zip files are still detected. In the [source code](#), there's a recipe for generating zbsm.extra.zip, which is like zbsm.zip except that it uses [extra-field quoting](#) instead of [DEFLATE quoting](#)—if you are a customer of an AV service that detects zbsm.zip but not zbsm.extra.zip, you should ask for an explanation. Another simple variant is [inserting spacer files between the bomb files](#), which may fool certain overlap-detection algorithms.

Twitter user [@TVQQAAMAAAAEAAA](#) [reports](#) "McAfee AV on my test machine just exploded." I haven't independently confirmed it, nor do I have details such as a version number.

Tavis Ormandy [points out](#) that there are a number of "Timeout" results in [the VirusTotal for zblg.zip \(screenshot 2019-07-06\)](#). AhnLab-V3, ClamAV, DrWeb, Endgame, F-Secure, GData, K7AntiVirus, K7GW, MaxSecure, McAfee, McAfee-GW-Edition, Panda, Qihoo-360, Sophos ML, VBA32. [The results for zbsm.zip \(screenshot 2019-07-06\)](#) are similar, though with a different set of timed-out engines: Baido, Bkav, ClamAV, CMC, DrWeb, Endgame, ESET-NOD32, F-Secure, GData, Kingsoft, McAfee-GW-Edition, NANO-Antivirus, Acronis. Interestingly, there are no timeouts in [the results for zbxl.zip; \(screenshot 2019-07-06\)](#) perhaps this means that some antivirus doesn't support Zip64?

Forum user 100 [reported](#) that a certain ESET product did not detect zbxl.zip, possibly because it uses Zip64. An update in the thread three days later showed the product being updated to detect it.

In [ClamAV bug 12356](#), Hanno Böck reported that zblg.zip caused high CPU usage in clamscan. [An initial patch](#) to detect overlapping files [turned out to be incomplete](#) because it only checked adjacent pairs of files. (I personally mishandled this issue by posting details of a workaround on the bug tracker, instead of reporting it privately.) [A later patch](#) imposed a time limit on file analysis.

2020-07-28: FlyTech Videos presented a [video testing various zip bombs](#), including [zbxl.zip](#), against Windows Defender, Windows Explorer, and 7-zip.

In my web server logs, I noticed a number of referers that appear to point to bug trackers.



- <http://jira.athr.ru/browse/WEB-12882>
- <https://project.avira.org/browse/ENGINE-2307>
- <https://project.avira.org/browse/ENGINE-2363>
- <https://topdesk-imp.cicapp.nl/tas/secure/mango/window/4>
- <https://jira-eng-rtp3.cisco.com/jira/browse/AMP4E-4849>
- <https://jira-eng-sjc1.cisco.com/jira/browse/CLAM-965>
- <https://flightdataservices.atlassian.net/secure/RapidBoard.jspa?selectedIssue=FDS-136>
- <https://projects.ucd.gpn.gov.uk/browse/VULN-1483>
- <https://testrail-int.qa1.immunet.com/index.php?cases/view/923720>
- <http://redmine-int-prod.intranet.cnim.net/issues/5596>
- <https://bugs.drweb.com/view.php?id=159759>
- <https://dev-jira.dynatrace.org/browse/APM-188227>
- <https://webgate.ec.europa.eu/CITnet/jira/browse/EPREL-2150>
- <https://jira.egnyte-it.com/browse/IN-8480>
- <https://jira.hq.eset.com/browse/CCDBL-1492>
- [https://bugzilla.olympus.f5net.com/show\\_bug.cgi?id=819053](https://bugzilla.olympus.f5net.com/show_bug.cgi?id=819053)
- [https://mantis.fortinet.com/bug\\_view\\_page.php?bug\\_id=0570222](https://mantis.fortinet.com/bug_view_page.php?bug_id=0570222)
- <https://redmine.joesecurity.org:64998/issues/4705>
- <http://dev.maildev.jp/mantis/view.php?id=5839>
- <https://confluence.managed.lu/pages/viewpage.action?pageId=47974242>
- <https://jira-lvs.prod.mcafee.com/browse/TSWS-653>
- <https://jira.modulbank.ru/browse/PV-33012>
- <http://jira.netzwerk.intern:8080/browse/SALES-81>
- <https://jira-hq.paloaltonetworks.local/browse/CON-43391>
- <https://jira-hq.paloaltonetworks.local/browse/GSRT-11680>
- <https://jira-hq.paloaltonetworks.local/browse/PAN-124201>
- <https://paynearme.atlassian.net/browse/PNM-4494>
- <https://jira.proofpoint.com/jira/browse/PE-29410>
- <https://dev.pulsesecure.net/jira/browse/PRS-379163>
- <https://qualtrics.atlassian.net/browse/APP-326>
- <https://jira.sastdev.net/browse/CIS-2819>
- <https://jira.sastdev.net/secure/RapidBoard.jspa?selectedIssue=EC-709>
- [https://bugzilla.seeburger.de/show\\_bug.cgi?id=89294](https://bugzilla.seeburger.de/show_bug.cgi?id=89294)
- [https://svm.cert.siemens.com/auseno/create\\_edit\\_vulnerability.php?vulnid=48573](https://svm.cert.siemens.com/auseno/create_edit_vulnerability.php?vulnid=48573)
- <https://jira.sophos.net/browse/CPISSUE-6560>
- <https://jira.vrt.sourcefire.com/browse/TT-1070>
- <https://task.jarvis.trendmicro.com/browse/JPSE-10432>
- <https://segjira.trendmicro.com:8443/browse/SEG-55636>
- <https://segjira.trendmicro.com:8443/browse/SEG-58824>
- <https://ucsc-cgl.atlassian.net/secure/RapidBoard.jspa?selectedIssue=SEAB-327>
- <https://jira.withbc.com/browse/BC-43950>
- <https://zscaler.zendesk.com/agent/tickets/849971>

## web browsers

I didn't directly experience this myself, but reports online say that Chrome and Safari may automatically unzip files after downloading.

- [ittakir](#): "Скачал самый маленький файл на 5GB, Chrome тут же начал его распаковывать, хотя его об этом не просили, ну и кушать процессор и диск." *"I downloaded the smallest file on 5GB, Chrome immediately began to unpack it, although it was not asked for it, well, to eat the processor and disk."*
- [Rzah](#): "Yet another reason why 'Open Safe files after downloading' is a stupid default setting for a web browser."

Chromium commit [f04d9b15bd1cba1433ad5453bc3ebff933d0e3bb](#) is perhaps related:

Add metrics detecting anomalously high ZIP compression ratios

It's possible for a single ZIP entry to be very large, even if we only scan small ZIP archives. These metrics will measure how often that occurs.

## filesystems

Something I didn't anticipate: unzipping one of the bombs on a compressed filesystem can be relatively safe.

- [flying\\_gel](#): "If I unzip this onto a compressed zfs dataset, will the resulting file be small? Edit: Just did a small test with a 42KB->5.5GB zip bomb. I ended up with 165MB worth of files so while just 3% of the full bomb, it's still a 4028 times inflation. ... I only have the standard LZ4 compression enabled, no dedup."

## Twitter

Links to this article had been widely shared on Twitter since around 2019-07-02, but around 2019-07-20 it began showing an ["unsafe link" interstitial](#) ([screenshot](#), [archive](#)).

## Safe Browsing

Sometime around 2019-07-23 it seems that this page, and *every* page on a \*.bamsoftware.com domain, got added to the [Safe Browsing](#) service used by web browsers to block malware and phishing sites. [Site status check](#), [block page screenshot](#). From a few quick checks, it looks like pages on bamsoftware.com have been demoted or delisted on the google.com search engine as well.

The Safe Browsing block is a bit annoying, because it disrupted [Snowflake](#), a completely unrelated service that happened to use the domain snowflake-broker.bamsoftware.com, which did not even host any files but was strictly a web API server. See [#31230 Firefox addon blocked from agent by Google Safe Browsing service](#).

The Safe Browsing block seemed to end on or before [2019-08-16](#).

## Xfinity xFi Protected Browsing

On 2019-11-26, I was informed by Hooman Mohajeri Moghaddam that the Comcast Xfinity xFi ["Protected Browsing"](#) feature blocks the bamsoftware.com domain, including this page ([screenshot](#)).

## D std.zip

The D programming language [made a modification](#) to the [std.zip module](#) to detect overlapping files.

## Apple iOS and iPadOS

Dzmitry Plotnikau sent me a report saying that a zip bomb could use up all cache storage on iPhones running iOS 12 and 13, even if only opened using "Quick look." The exhaustion of storage could have various side effects, including misbehaving apps, deletion of local cloud files, and OS crashes, in some cases requiring a factory reset to remedy. The bug was mitigated in iOS 14.0 (and likely other, contemporaneous point release of iOS and iPadOS). See [HT211850](#) under the "libarchive" heading.

# A final plea

It's time to put an end to Facebook. Working there is not ethically neutral: every day that you go into work, you are doing something wrong. If you have a Facebook account, delete it. If you work at Facebook, quit.

And let us not forget that the National Security Agency must be destroyed.