Exploiting aCropalypse: Recovering Truncated PNGs

By David Buchanan, 18th of March 2023

This article assumes you've already heard about the aCropalypse vulnerability, aka CVE-2023-21036. If not, go read about it **here** (oops, this page doesn't exist yet. Read this **tweet** in the meantime).

For me, it all started with the following conversation:

> **Simon** — 2023-01-02 15:16
>
> can you decompress a zlib stream which is missing the first say 0x10000 bytes
> but you still have 0x100000 bytes of the data trailing it
> is that possible

> **nicolás** — 2023-01-02 15:22
>
> I doubt it because you could have gigabytes of data that all depend on data referenced from the first few bytes
> It might be doable if you have the corresponding uncompressed data but you're missing the zlib stream
> Sounds like a Retr0id task either way

> **Simon** — 2023-01-02 15:24
>
> if it is possible i've found a massive security flaw
>
> **Retr0id** — 2023-01-02 15:27
>
> possible but hard
> there's no easy way to re-sync a truncated zlib stream
>
> **Simon** — 2023-01-02 15:27
>
> can i dm you about this
>
> **Retr0id** — 2023-01-02 15:27
>
> sure

At this point in time, Simon was asking the question in an abstract sense, so as not to give away the nature of the vulnerability.

Nicolás' and my initial responses were both correct, in the *general* case.

Zlib uses DEFLATE compression, which itself makes use of **LZ77** and **Huffman Coding** techniques. Nicolas' answer is based on the fact that LZ77 works by replacing repeated data with backreferences to a previous occurrence. If you're missing that prior data, then you'll never be able to resolve backreferences to it.

My answer was based on the fact that zlib streams can use dynamic Huffman coding, where a custom Huffman tree is defined at the start of a block. This tree is used to encode symbols for the rest of that block. It's near-impossible to decompress Huffman-coded data without knowing the tree used—almost like trying to decrypt an encrypted message without knowing the key.

These two issues make it sound like an impossible problem to solve in the general case, and that's true! However, we're not dealing with the general case here—

there’s a specific scenario. You already know the scenario because you read the title of this post, but I found out when Simon explained it to me privately:

> **SimonTime** — 2023-01-02 15:28
>
> so basically the pixel 7 pro, when you crop and save a screenshot, overwrites the image with the new version, but leaves the rest of the original file in its place
>
> **Retr0id** — 2023-01-02 15:28
>
> ohhhhhhh wow
>
> **SimonTime** — 2023-01-02 15:28
>
> so if you were to take a screenshot of an app which shows your address on screen, then crop it, if you could recover the information somehow that's a big deal
>
> **Retr0id** — 2023-01-02 15:29
>
> yeah that does sound pretty big
> so uh, I just started drinking a can of monster
> I am well positioned to start writing code to do this lol
>
> **SimonTime** — 2023-01-02 15:29
>
> i'll send you some samples

At this point in time, I had all sorts of cryptanalysis-adjacent ideas swirling around in my head. If we knew what some sub-section of the screenshot was supposed to look like (due to having the cropped version), that would act as a "known plaintext". By modelling Huffman coding as a cryptographic cipher, with the unknown Huffman tree as the key, we might be able to crack it.

I started analysing the sample screenshots using a janky zlib-parsing Python script I had lying around (I'm working on a low-level zlib manipulation tool, watch this space!). I found that the data was indeed compressed using dynamic Huffman coding, but that the Huffman tree was re-specified every ~16KB or so. This was *excellent* news because it meant I didn't have to solve the "Huffman coding cryptanalysis" problem, and could instead solve the much simpler-in-practice problem of "find the start of a dynamic-Huffman-coded zlib block". How do we do that?

The Algorithm

After all that complexity, the algorithm I came up with is deceptively simple. It can be summarised with the following pseudocode:

```
for each bit-offset:
    if it doesn't look like the start of a dynamic huffman block:
        skip this offset

    try decompressing from that offset:
        if the decompressed data looks plausible:
            return decompressed data!
    catch decompression errors:
        continue
```

We need to deal with bit-offsets as opposed to byte-offsets because zlib is a bit-oriented format, and a new block could start at any bit offset within a byte. Perhaps unsurprisingly, the zlib library does not have built-in support for starting from a specific bit offset. To solve this, my Python implementation precomputes a list of 8 bytestrings, each with the bits shifted by n bits. In a loop, those bytestrings are sliced up and passed into zlib as needed.

Another thing to consider is that zlib has a "window size" of up to 32KB. This means that compressed data can make backreferences to any of the previous 32K bytes - for example, "look back 1312 bytes and make a copy of the subsequent 42 bytes". If the stream we're decompressing tries to make backreferences to before the "start" of the stream, zlib will error out (i.e. the problem Nicolás was describing earlier). To work around this, I pre-initialised zlib's state with 32KB of the ascii character 'X'. This was an arbitrary choice, but it's convenient because it means that any of these "made up" byte values will show up as a uniform grey colour in the final PNG.

Theoretically, an image could be made up almost entirely of back-references to missing data, but *in practice* most images aren't like this. We can get an intelligible result most of the time, even if we are missing *some* back-referenced data.

There's a couple of PNG-related details still left to consider, primarily how to parse the zlib stream out of the truncated PNG, and how to turn the recovered decompressed data into something viewable. This is all fairly trivial stuff, if you've read my **"Hello, PNG!"** blog post recently.

After a few hours of fiddling around, I had a **working PoC**. By this time Simon was asleep (yay timezones), so I was left on my own to freak out over the implications of the code I'd just written:

> **Retr0id** — 2023-01-02 21:11
>
> holy fuck this is bad

If it were in my lexicon at the time, I might've said "holy fucking bingle".

Simon has ported my PoC to C and compiled it to WASM, so you can use it on the web **here** to test if your image files are vulnerable (all image processing is done client-side).

Personal Impact

Although I'm currently an iPhone user, I used to use a Pixel 3XL. I'm also a heavy Discord user, and in the past I'd shared plenty of cropped screenshots through the Discord app.

I wrote a script to scrape my own message history to look for vulnerable images. There were lots of them, although most didn't leak any particularly private information.

The worst instance was when I posted a cropped screenshot of an eBay order confirmation email, showing the product I'd just bought. Through the exploit, I was able to un-crop that screenshot, revealing my full postal address (which was also present in the email). That's pretty bad!

Root Cause Analysis

The bug lies in closed-source Google-proprietary code so it's a bit hard to inspect, but after some patch-diffing I concluded that the root cause was due to this horrible bit of API "design": **https://issuetracker.google.com/issues/180526528**.

Google was passing `"w"` to a call to `parseMode()`, when they should've been passing `"wt"` (the t stands for truncation). This is an easy mistake, since similar APIs (like POSIX **fopen**) will truncate by default when you simply pass `"w"`. Not only that, but previous Android releases had `parseMode("w")` truncate by default too! This change wasn't even documented until some time after the aforementioned **bug report** was made.

The end result is that the image file is opened *without* the `O_TRUNC` flag, so that when the cropped image is written, the original image is not truncated. If the new image file is smaller, the end of the original is left behind.

IMHO, the takeaway here is that API footguns should be treated as security vulnerabilities.
