# A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution

googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

Posted by Ian Beer & Samuel Groß of Google Project Zero

We want to thank Citizen Lab for sharing a sample of the FORCEDENTRY exploit with us, and Apple's Security Engineering and Architecture (SEAR) group for collaborating with us on the technical analysis. The editorial opinions reflected below are solely Project Zero's and do not necessarily reflect those of the organizations we collaborated with during this research.

Earlier this year, Citizen Lab managed to capture an NSO iMessage-based zero-click exploit being used to target a Saudi activist. In this two-part blog post series we will describe for the first time how an in-the-wild zero-click iMessage exploit works.

Based on our research and findings, we assess this to be one of the most technically sophisticated exploits we've ever seen, further demonstrating that the capabilities NSO provides rival those previously thought to be accessible to only a handful of nation states.

The vulnerability discussed in this blog post was fixed on September 13, 2021 in iOS 14.8 as CVE-2021-30860.

NSO

NSO Group is one of the highest-profile providers of "access-as-a-service", selling packaged hacking solutions which enable nation state actors without a home-grown offensive cyber capability to "pay-to-play", vastly expanding the number of nations with such cyber capabilities.

For years, groups like Citizen Lab and Amnesty International have been tracking the use of NSO's mobile spyware package "Pegasus". Despite NSO's claims that they "[evaluate] the potential for adverse human rights impacts arising from the misuse of NSO products" Pegasus has been linked to the hacking of the New York Times journalist Ben Hubbard by the Saudi regime, hacking of human rights defenders in Morocco and Bahrain, the targeting of Amnesty International staff and dozens of other cases.

Last month the United States added NSO to the "Entity List", severely restricting the ability of US companies to do business with NSO and stating in a press release that "[NSO's tools] enabled foreign governments to conduct transnational repression, which is the practice of authoritarian governments targeting dissidents, journalists and activists outside of their sovereign borders to silence dissent."

Citizen Lab was able to recover these Pegasus exploits from an iPhone and therefore this analysis covers NSO's capabilities against iPhone. We are aware that NSO sells similar zero-click capabilities which target Android devices; Project Zero does not have samples of these exploits but if you do, please reach out.

From One to Zero

In previous cases such as the Million Dollar Dissident from 2016, targets were sent links in SMS messages:


Screenshots of Phishing SMSs reported to Citizen Lab in 2016

source: https://citizenlab.ca/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/

The target was only hacked when they clicked the link, a technique known as a one-click exploit. Recently, however, it has been documented that NSO is offering their clients zero-click exploitation technology, where even very technically savvy targets who might not click a phishing link are completely unaware they are being targeted. In the zero-click scenario no user interaction is required. Meaning, the attacker doesn't need to send phishing messages; the exploit just works silently in the background. Short of not using a device, there is no way to prevent exploitation by a zero-click exploit; it's a weapon against which there is no defense.

One weird trick

The initial entry point for Pegasus on iPhone is iMessage. This means that a victim can be targeted just using their phone number or AppleID username.

iMessage has native support for GIF images, the typically small and low quality animated images popular in meme culture. You can send and receive GIFs in iMessage chats and they show up in the chat window. Apple wanted to make those GIFs loop endlessly rather than only play once, so very early on in the iMessage parsing and processing pipeline (after a message has been received but well before the message is shown), iMessage calls the following method in the IMTranscoderAgent process (outside the "BlastDoor" sandbox), passing any image file received with the extension .gif:

```
[IMGIFUtils copyGifFromPath:toDestinationPath:error]
```

Looking at the selector name, the intention here was probably to just copy the GIF file before editing the loop count field, but the semantics of this method are different. Under the hood it uses the CoreGraphics APIs to render the source image to a new GIF file at the destination path. And just because the source filename has to end in .gif, that doesn't mean it's really a GIF file.

The ImageIO library, as detailed in a previous Project Zero blogpost, is used to guess the correct format of the source file and parse it, completely ignoring the file extension. Using this "fake gif" trick, over 20 image codecs are suddenly part of the iMessage zero-click attack surface, including some very obscure and complex formats, remotely exposing probably hundreds of thousands of lines of code.

Note: Apple inform us that they have restricted the available ImageIO formats reachable from IMTranscoderAgent starting in iOS 14.8.1 (26 October 2021), and completely removed the GIF code path from IMTranscoderAgent starting in iOS 15.0 (20 September 2021), with GIF decoding taking place entirely within BlastDoor.

A PDF in your GIF

NSO uses the "fake gif" trick to target a vulnerability in the CoreGraphics PDF parser.

PDF was a popular target for exploitation around a decade ago, due to its ubiquity and complexity. Plus, the availability of javascript inside PDFs made development of reliable exploits far easier. The CoreGraphics PDF parser doesn't seem to interpret javascript, but NSO managed to find something equally powerful inside the CoreGraphics PDF parser...

Extreme compression

In the late 1990's, bandwidth and storage were much more scarce than they are now. It was in that environment that the JBIG2 standard emerged. JBIG2 is a domain specific image codec designed to compress images where pixels can only be black or white.

It was developed to achieve extremely high compression ratios for scans of text documents and was implemented and used in high-end office scanner/printer devices like the XEROX WorkCenter device shown below. If you used the scan to pdf functionality of a device like this

a decade ago, your PDF likely had a JBIG2 stream in it.


A Xerox WorkCentre 7500 series multifunction printer, which used JBIG2

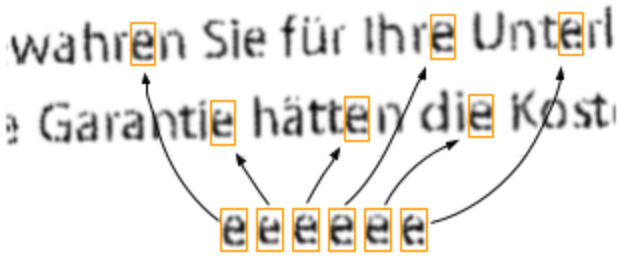
for its scan-to-pdf functionality

source: https://www.office.xerox.com/en-us/multifunction-printers/workcentre-7545-7556/specifications

The PDFs files produced by those scanners were exceptionally small, perhaps only a few kilobytes. There are two novel techniques which JBIG2 uses to achieve these extreme compression ratios which are relevant to this exploit:

Technique 1: Segmentation and substitution

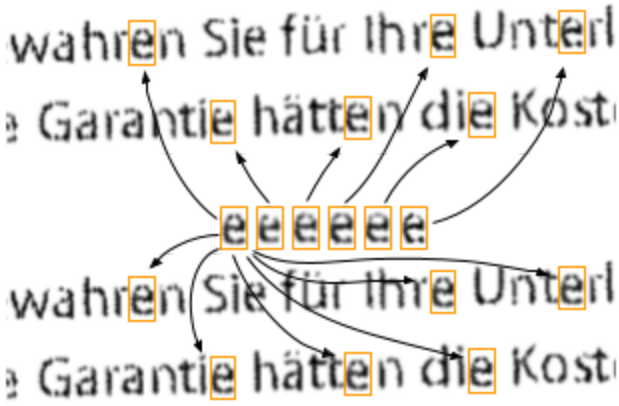
Effectively every text document, especially those written in languages with small alphabets like English or German, consists of many repeated letters (also known as glyphs) on each page. JBIG2 tries to segment each page into glyphs then uses simple pattern matching to match up glyphs which look the same:




Simple pattern matching can find all the shapes which look similar on a page,

in this case all the 'e's

JBIG2 doesn't actually know anything about glyphs and it isn't doing OCR (optical character recognition.) A JBIG encoder is just looking for connected regions of pixels and grouping similar looking regions together. The compression algorithm is to simply substitute all sufficiently-similar looking regions with a copy of just one of them:




Replacing all occurrences of similar glyphs with a copy of just one often yields a document which is still quite legible and enables very high compression ratios

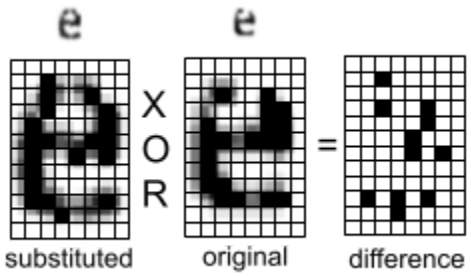
In this case the output is perfectly readable but the amount of information to be stored is significantly reduced. Rather than needing to store all the original pixel information for the whole page you only need a compressed version of the "reference glyph" for each character and the relative coordinates of all the places where copies should be made. The decompression algorithm then treats the output page like a canvas and "draws" the exact same glyph at all the stored locations.

There's a significant issue with such a scheme: it's far too easy for a poor encoder to accidentally swap similar looking characters, and this can happen with interesting consequences. D. Kriesel's blog has some motivating examples where PDFs of scanned invoices have different figures or PDFs of scanned construction drawings end up with incorrect measurements. These aren't the issues we're looking at, but they are one significant reason why JBIG2 is not a common compression format anymore.

Technique 2: Refinement coding

As mentioned above, the substitution based compression output is lossy. After a round of compression and decompression the rendered output doesn't look exactly like the input. But JBIG2 also supports lossless compression as well as an intermediate "less lossy" compression mode.

It does this by also storing (and compressing) the difference between the substituted glyph and each original glyph. Here's an example showing a difference mask between a substituted character on the left and the original lossless character in the middle:




Using the XOR operator on bitmaps to compute a difference image

In this simple example the encoder can store the difference mask shown on the right, then during decompression the difference mask can be XORed with the substituted character to recover the exact pixels making up the original character. There are some more tricks outside of the scope of this blog post to further compress that difference mask using the intermediate forms of the substituted character as a "context" for the compression.

Rather than completely encoding the entire difference in one go, it can be done in steps, with each iteration using a logical operator (one of AND, OR, XOR or XNOR) to set, clear or flip bits. Each successive refinement step brings the rendered output closer to the original and this allows a level of control over the "lossiness" of the compression. The implementation of these refinement coding steps is very flexible and they are also able to "read" values already present on the output canvas.

A JBIG2 stream

Most of the CoreGraphics PDF decoder appears to be Apple proprietary code, but the JBIG2 implementation is from Xpdf, the source code for which is freely available.

The JBIG2 format is a series of segments, which can be thought of as a series of drawing commands which are executed sequentially in a single pass. The CoreGraphics JBIG2 parser supports 19 different segment types which include operations like defining a new page, decoding a huffman table or rendering a bitmap to given coordinates on the page.

Segments are represented by the class JBIG2Segment and its subclasses JBIG2Bitmap and JBIG2SymbolDict.

A JBIG2Bitmap represents a rectangular array of pixels. Its data field points to a backing-buffer containing the rendering canvas.

A JBIG2SymbolDict groups JBIG2Bitmaps together. The destination page is represented as a JBIG2Bitmap, as are individual glyphs.

JBIG2Segments can be referred to by a segment number and the GList vector type stores pointers to all the JBIG2Segments. To look up a segment by segment number the GList is scanned sequentially.

The vulnerability

The vulnerability is a classic integer overflow when collating referenced segments:

```
Guint numSyms; // (1)
numSyms = 0;
for (i = 0; i < nRefSegs; ++i) {
  if ((seg = findSegment(refSegs[i]))) {
    if (seg->getType() == jbig2SegSymbolDict) {
      numSyms += ((JBIG2SymbolDict *)seg)->getSize();  // (2)
    } else if (seg->getType() == jbig2SegCodeTable) {
      codeTables->append(seg);
    }
  } else {
    error(errSyntaxError, getPos(),
          "Invalid segment reference in JBIG2 text region");
    delete codeTables;
    return;
  }
```

```
  }
...
  // get the symbol bitmaps
  syms = (JBIG2Bitmap **)gmallocn(numSyms, sizeof(JBIG2Bitmap *)); // (3)
  kk = 0;
  for (i = 0; i < nRefSegs; ++i) {
    if ((seg = findSegment(refSegs[i]))) {
      if (seg->getType() == jbig2SegSymbolDict) {
        symbolDict = (JBIG2SymbolDict *)seg;
        for (k = 0; k < symbolDict->getSize(); ++k) {
          syms[kk++] = symbolDict->getBitmap(k); // (4)
        }
      }
    }
  }
```

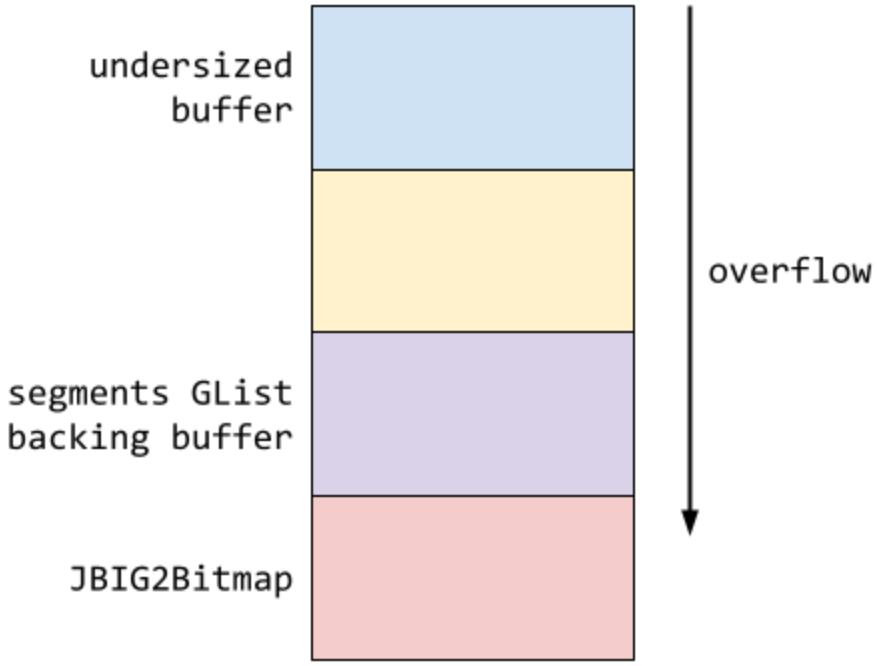
numSyms is a 32-bit integer declared at (1). By supplying carefully crafted reference segments it's possible for the repeated addition at (2) to cause numSyms to overflow to a controlled, small value.

That smaller value is used for the heap allocation size at (3) meaning syms points to an undersized buffer.

Inside the inner-most loop at (4) JBIG2Bitmap pointer values are written into the undersized syms buffer.

Without another trick this loop would write over 32GB of data into the undersized syms buffer, certainly causing a crash. To avoid that crash the heap is groomed such that the first few writes off of the end of the syms buffer corrupt the GList backing buffer. This GList stores all known segments and is used by the findSegments routine to map from the segment numbers passed in refSegs to JBIG2Segment pointers. The overflow causes the JBIG2Segment pointers in the GList to be overwritten with JBIG2Bitmap pointers at (4).

Conveniently since JBIG2Bitmap inherits from JBIG2Segment the seg->getType() virtual call succeed even on devices where Pointer Authentication is enabled (which is used to perform a weak type check on virtual calls) but the returned type will now not be equal to jbig2SegSymbolDict thus causing further writes at (4) to not be reached and bounding the extent of the memory corruption.

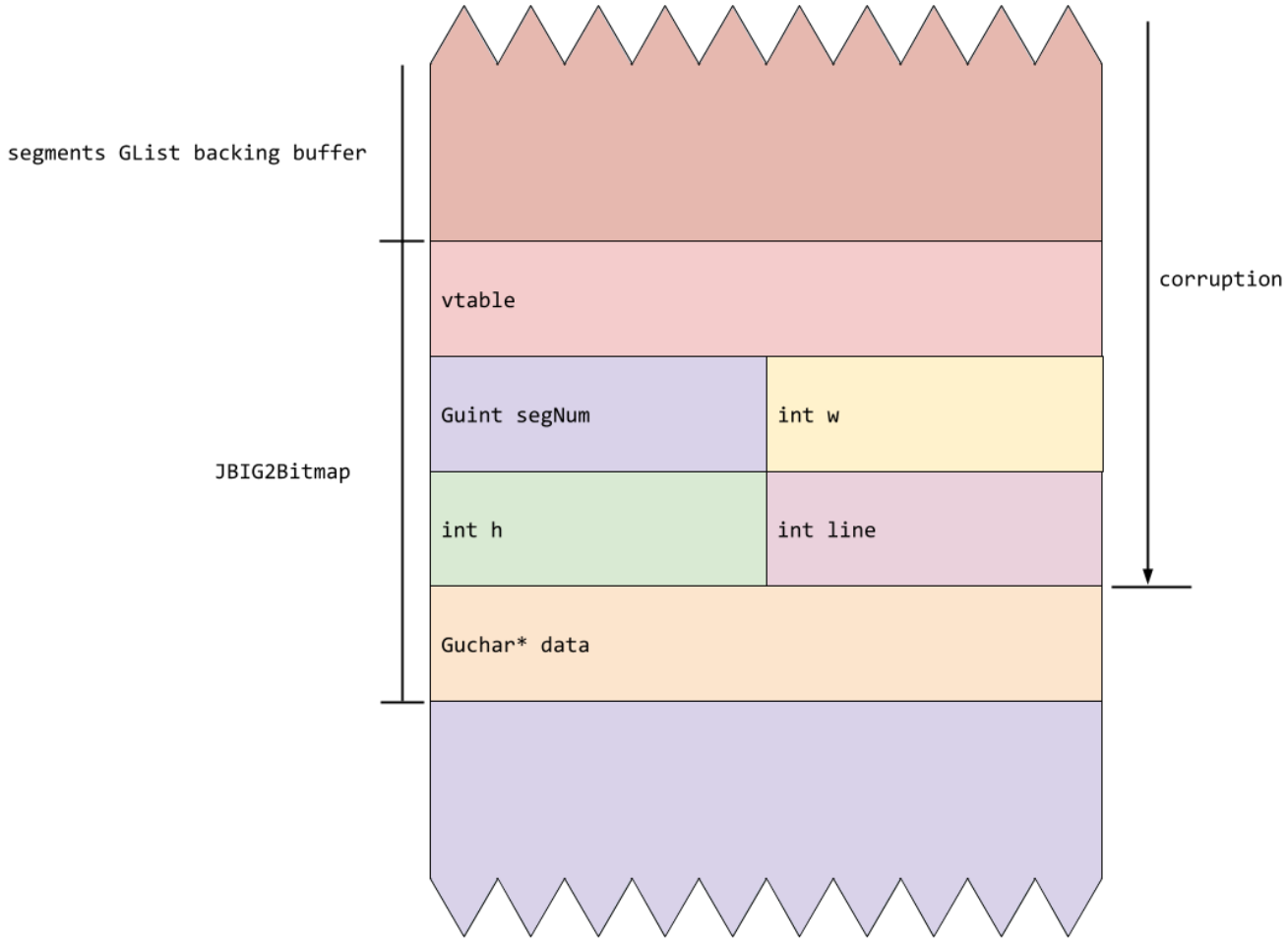



A simplified view of the memory layout when the heap overflow occurs showing the undersized-buffer below the GList backing buffer and the JBIG2Bitmap

Boundless unbounding

Directly after the corrupted segments GList, the attacker grooms the JBIG2Bitmap object which represents the current page (the place to where current drawing commands render).

JBIG2Bitmaps are simple wrappers around a backing buffer, storing the buffer's width and height (in bits) as well as a line value which defines how many bytes are stored for each line.

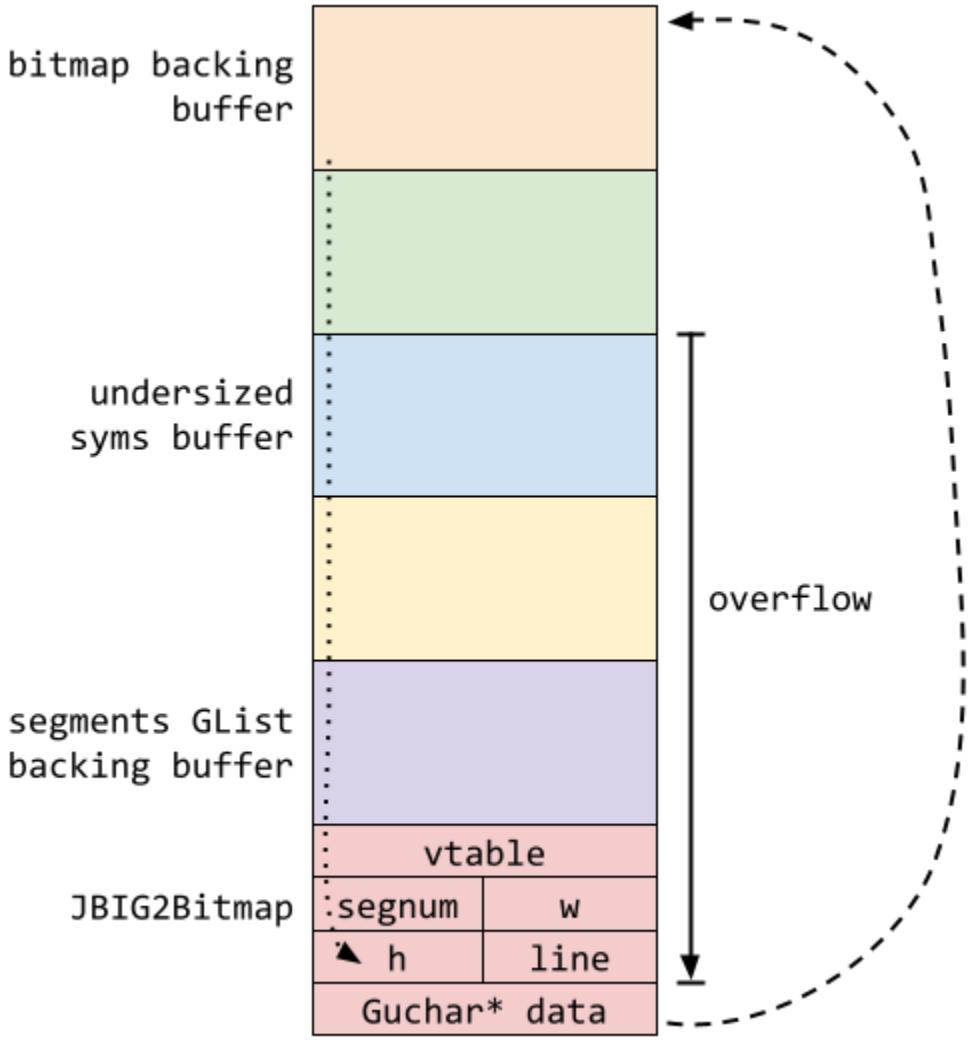



The memory layout of the JBIG2Bitmap object showing the segnum, w, h and line fields which are corrupted during the overflow

By carefully structuring refSegs they can stop the overflow after writing exactly three more JBIG2Bitmap pointers after the end of the segments GList buffer. This overwrites the vtable pointer and the first four fields of the JBIG2Bitmap representing the current page. Due to the nature of the iOS address space layout these pointers are very likely to be in the second 4GB of virtual memory, with addresses between 0x100000000 and 0x1fffffff. Since all iOS hardware is little endian (meaning that the w and line fields are likely to be overwritten with 0x1 — the most-significant half of a JBIG2Bitmap pointer) and the segNum and h fields are likely to be overwritten with the least-significant half of such a pointer, a fairly random value depending on heap layout and ASLR somewhere between 0x100000 and 0xffffffff.

This gives the current destination page JBIG2Bitmap an unknown, but very large, value for h. Since that h value is used for bounds checking and is supposed to reflect the allocated size of the page backing buffer, this has the effect of "unbounding" the drawing canvas. This means that subsequent JBIG2 segment commands can read and write memory outside of the original bounds of the page backing buffer.

The heap groom also places the current page's backing buffer just below the undersized syms buffer, such that when the page JBIG2Bitmap is unbounded, it's able to read and write its own fields:




The memory layout showing how the unbounded bitmap backing buffer is able to reference the JBIG2Bitmap object and modify fields in it as it is located after the backing buffer in memory

By rendering 4-byte bitmaps at the correct canvas coordinates they can write to all the fields of the page JBIG2Bitmap and by carefully choosing new values for w, h and line, they can write to arbitrary offsets from the page backing buffer.

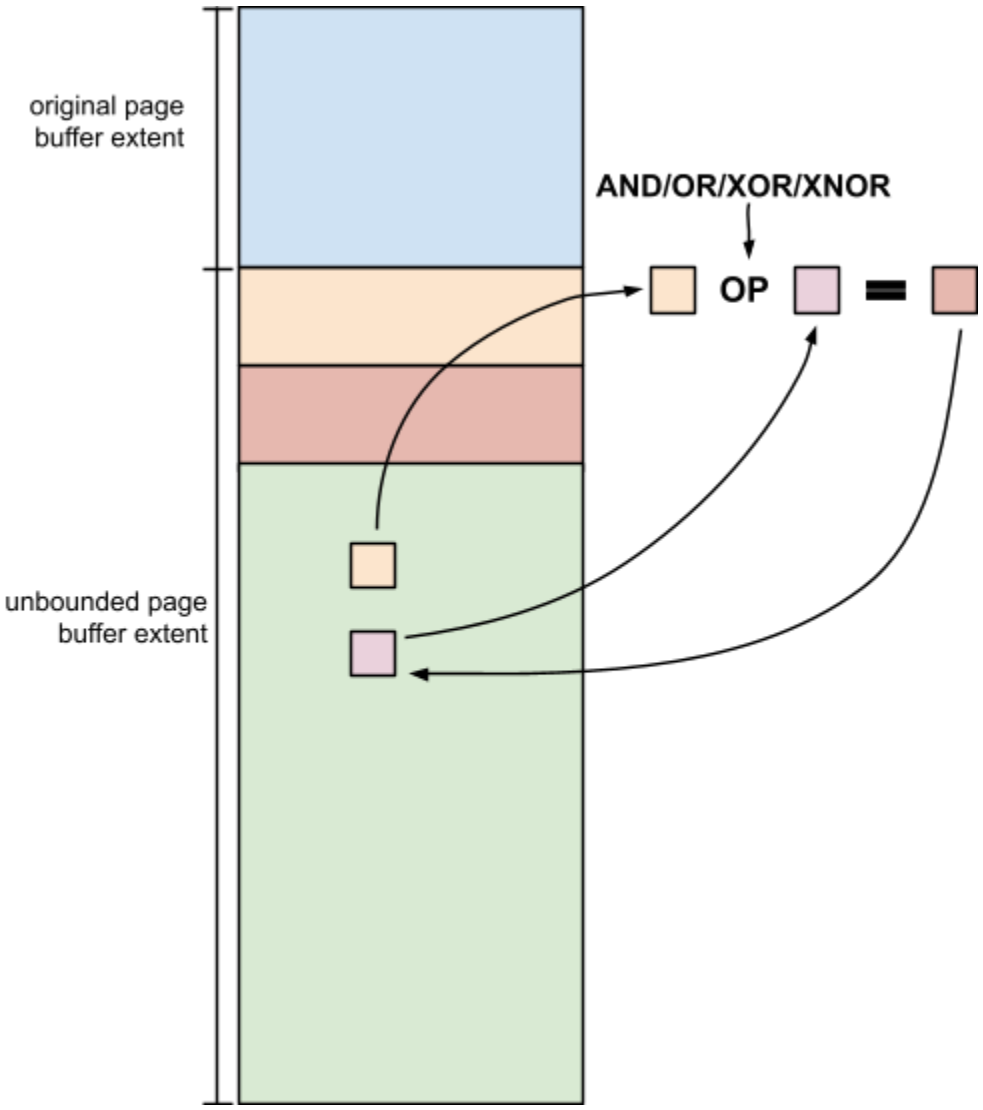
At this point it would also be possible to write to arbitrary absolute memory addresses if you knew their offsets from the page backing buffer. But how to compute those offsets? Thus far, this exploit has proceeded in a manner very similar to a "canonical" scripting language exploit which in Javascript might end up with an unbounded ArrayBuffer object with access

to memory. But in those cases the attacker has the ability to run arbitrary Javascript which can obviously be used to compute offsets and perform arbitrary computations. How do you do that in a single-pass image parser?

My other compression format is turing-complete!

As mentioned earlier, the sequence of steps which implement JBIG2 refinement are very flexible. Refinement steps can reference both the output bitmap and any previously created segments, as well as render output to either the current page or a segment. By carefully crafting the context-dependent part of the refinement decompression, it's possible to craft sequences of segments where only the refinement combination operators have any effect.

In practice this means it is possible to apply the AND, OR, XOR and XNOR logical operators between memory regions at arbitrary offsets from the current page's JBIG2Bitmap backing buffer. And since that has been unbounded… it's possible to perform those logical operations on memory at arbitrary out-of-bounds offsets:

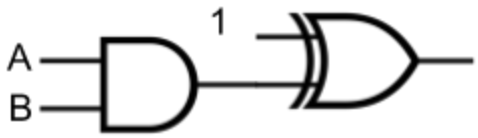



The memory layout showing how logical operators can be applied out-of-bounds

It's when you take this to its most extreme form that things start to get really interesting. What if rather than operating on glyph-sized sub-rectangles you instead operated on single bits?

You can now provide as input a sequence of JBIG2 segment commands which implement a sequence of logical bit operations to apply to the page. And since the page buffer has been unbounded those bit operations can operate on arbitrary memory.

With a bit of back-of-the-envelope scribbling you can convince yourself that with just the available AND, OR, XOR and XNOR logical operators you can in fact compute any computable function - the simplest proof being that you can create a logical NOT operator by XORing with 1 and then putting an AND gate in front of that to form a NAND gate:




An AND gate connected to one input of an XOR gate. The other XOR gate input is connected to the constant value 1 creating an NAND.

A NAND gate is an example of a universal logic gate; one from which all other gates can be built and from which a circuit can be built to compute any computable function.

Practical circuits

JBIG2 doesn't have scripting capabilities, but when combined with a vulnerability, it does have the ability to emulate circuits of arbitrary logic gates operating on arbitrary memory. So why not just use that to build your own computer architecture and script that!? That's exactly what this exploit does. Using over 70,000 segment commands defining logical bit operations, they define a small computer architecture with features such as registers and a full 64-bit adder and comparator which they use to search memory and perform arithmetic operations. It's not as fast as Javascript, but it's fundamentally computationally equivalent.

The bootstrapping operations for the sandbox escape exploit are written to run on this logic circuit and the whole thing runs in this weird, emulated environment created out of a single decompression pass through a JBIG2 stream. It's pretty incredible, and at the same time, pretty terrifying.

In a future post (currently being finished), we'll take a look at exactly how they escape the IMTranscoderAgent sandbox.