The Patch

objective-see.com/blog/blog_0x67.html

Analysis of CVE-2021-30860

the flaw and fix of a zero-click vulnerability, exploited in the wild

by: Tom McGuire / September 16, 2021

This guest blog post, was written by Tom McGuire, a senior instructor and cybersecurity focus area coordinator at Johns Hopkins and tech editor of my upcoming The Art of Mac Malware: Analysis book.

Here, he shares his analysis of reversing Apple's patch for CVE-2021-30860 (a zero-click iOS/macOS vulnerability exploited in the wild) ...highlighting both the underlying flaw, and Apple's fix.

Mahalo for sharing Tom! 🤩

ℹ️ For another write-up on this bug, see Mickey Jin's excellent post:

"Analyzing The ForcedEntry Zero-Click iPhone Exploit Used By Pegasus."

Wild, wild west - Quick Initial Analysis of CVE-2021-30860

Recently, Apple released iOS/iPadOS 14.8 and macOS Big Sur 11.6 which fixes both an integer overflow and a use after free vulnerability (the watchOS platform was also patched to fix the integer overflow issue). This blog post will analyze the integer overflow in CoreGraphics, `CVE-2021-30860`. After examining the modified .dylib, it appears that there were other issues that were resolved as well, related to imaging processing. We will focus in on the `JBIG2` processing, specifically in the `JBIG2::readTextRegionSeg`.

I could not find information about Apple's use of `JBIG2` libraries. However, as we will see there is a likely chance there was some collaboration with open source software (see: https://gitlab.freedesktop.org/poppler/poppler/-/blob/master/poppler/JBIG2Stream.cc). The source code shown is from poppler, but as shown in the header file the origin is “Copyright 2002-2003 Glyph & Cog, LLC”.

An integer overflow can lead to a variety of issues. A common result with an integer overflow is to cause a dynamic memory allocation (e.g. malloc(), calloc() etc..) to be too small. Later, data is copied from a source that is larger than the allocated size, resulting in a heap buffer overflow. (Not all integer overflows will manifest this way, but it is a common occurrence and relevant to this discussion.)

`CVE-2021-30860` is an integer overflow in the CoreGraphics component, specifically the decoding of a `JBIG2` data. `JBIG2` (Joint Bi-level Image Experts Group) is an image compression format which can be embedded as a stream in a PDF or PSD document, or potentially other formats as well. You can read more about it here.

Before we dive into the assembly and uncover the vulnerability and how it was fixed, we want to look at the discovery. CitizenLab reported this vulnerability, which they dubbed FORCEDENTRY (a knock at Apple's recent security component, BlastDoor!), to Apple after they had done some analysis on journalist's phones suspected of being hacked. In their reporting, CitizenLab attributes the attacks to the NSO group, due to the Pegasus software that was seen on these infected devices:

From Pearl to Pegasus

Bahraini Government Hacks Activists with NSO Group Zero-Click iPhone Exploits

By Bill Marczak, Ali Abdulemam[1], Noura Al-Jizawi, Siena Anstis, Kristin Berdan, John Scott-Railton, and Ron Deibert

[1] Red Line for Gulf

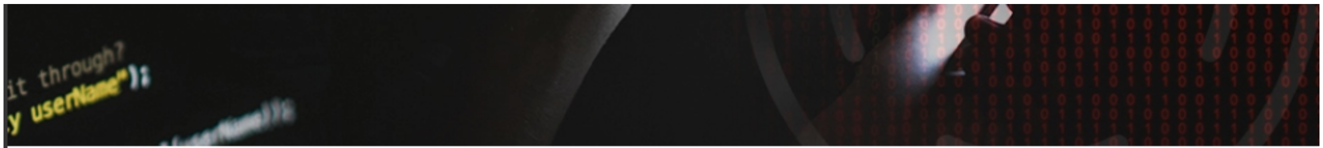
August 24, 2021

CitizenLab thoughts on Pegaus

During their analysis, they uncovered crash logs and noticed quite a few image files that seemed to crash the `IMTranscoderAgent`. `IMTranscoderAgent` is one of the components related to processing of iMessage data, including upon sending/receiving images!

According to CitizenLab, they reported the vulnerability to Apple on Tuesday, September 7, 2021 and Apple confirmed and released the patches for the issue on Monday, September 13, 2021. That is a quick turnaround, so let's see how well they did with the patching!

Image file formats are notorious for having vulnerabilities that can lead to arbitrary remote code execution (RCE) on devices (CVE-2009-1858, CVE-2015-6778, CVE-2020-1910, etc..).



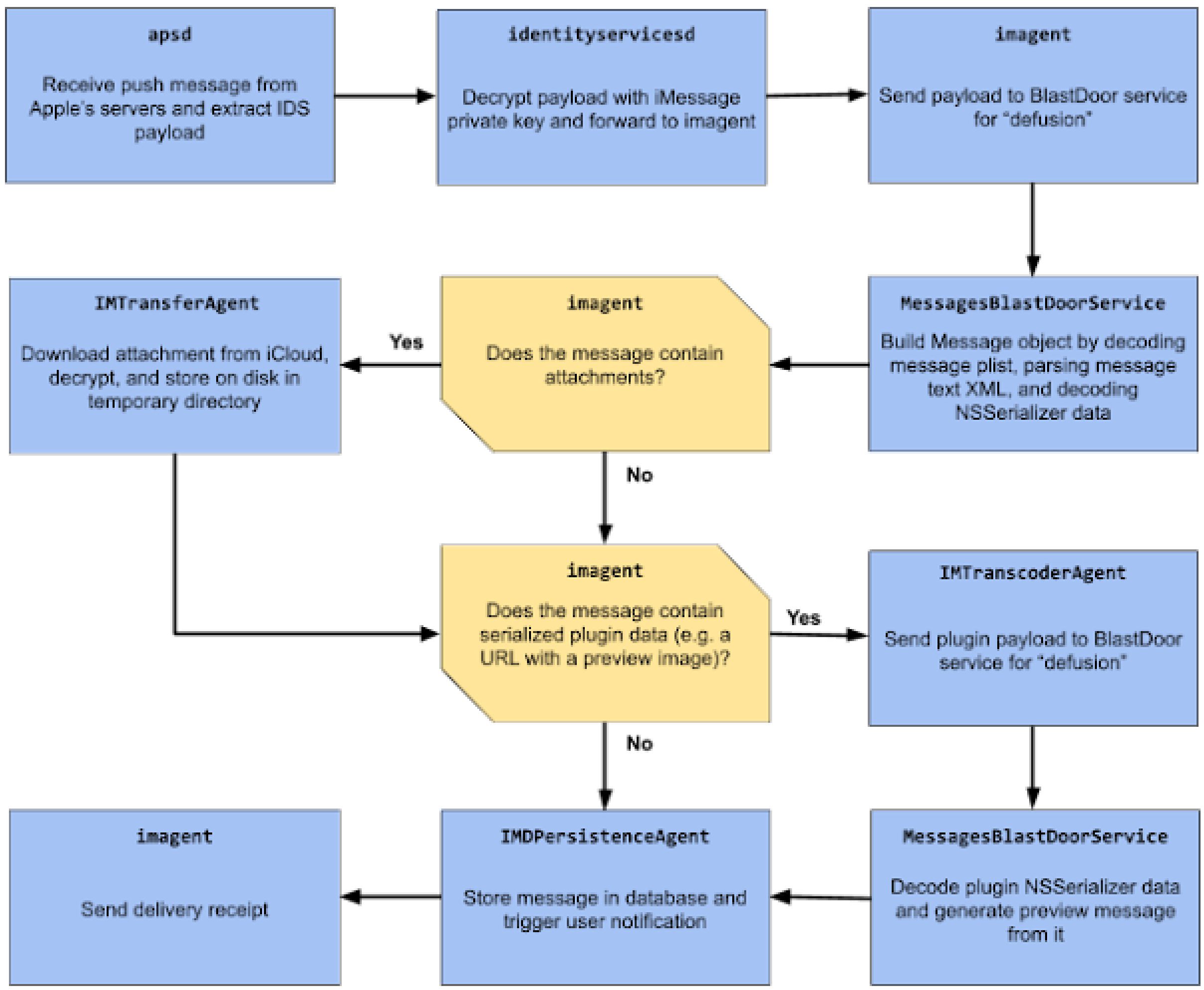
Now Patched Vulnerability in WhatsApp could have led to data exposure of users

September 2, 2021

Imaging parsing issues are not new!
It is not surprising that such an issue existed here. With this `JBIG2` processing vulnerability (which exists in the `readTextRegionSeg` method), I will note that another *very* similar vulnerability was previously patched. This issue is nearly the same logic as the one in FORCEDENTRY. The method `readSymbolDictSeg` contains integer overflow checks that help prevent the scenario that we will examine in this post! (Don't worry, we will get back to this and do a quick look to see this in assembly).

Of particular note to the attacks reported by CitizenLab, the file formats were PDF files, with embedded `JBIG2` streams. Zero-click iMessage vulnerabilities have existed before (see, here and here).

In an effort to help reduce this attack surface, Apple recently (iOS14) introduced the "BlastDoor" feature. Samuel Groß, of Google's Project 0, posted an excellent write-up about this new feature:



BlastDoor analysis by Google P0

For our purposes, what we need to understand is that the BlastDoor feature is meant to “sandbox” processing in the iMessage chain. In other words, when an image or document is received via iMessage and automatically parsed, it is done in a sandboxed environment. The intent is that, if a vulnerability exists in some of the processing engine, the exploitation will be limited to this sandboxed environment, keeping the rest of the system ‘safe’. This is true for certain file formats, but it appears that Apple did not sandbox all potential automatically parsed formats (looking at you PSD files, and likely other raster formats).

Though I have not gone through and analyzed any changes to BlastDoor since this patch, I can only hope that Apple has increased the robustness of BlastDoor and has prevented PDF, PSD and other raster file format parsing from going through the `IMTranscoderAgent`. That is, going forward, the hope is these other notoriously prone formats are processed in the BlastDoor sandboxed environment…perhaps we can look at that in a future blog post!

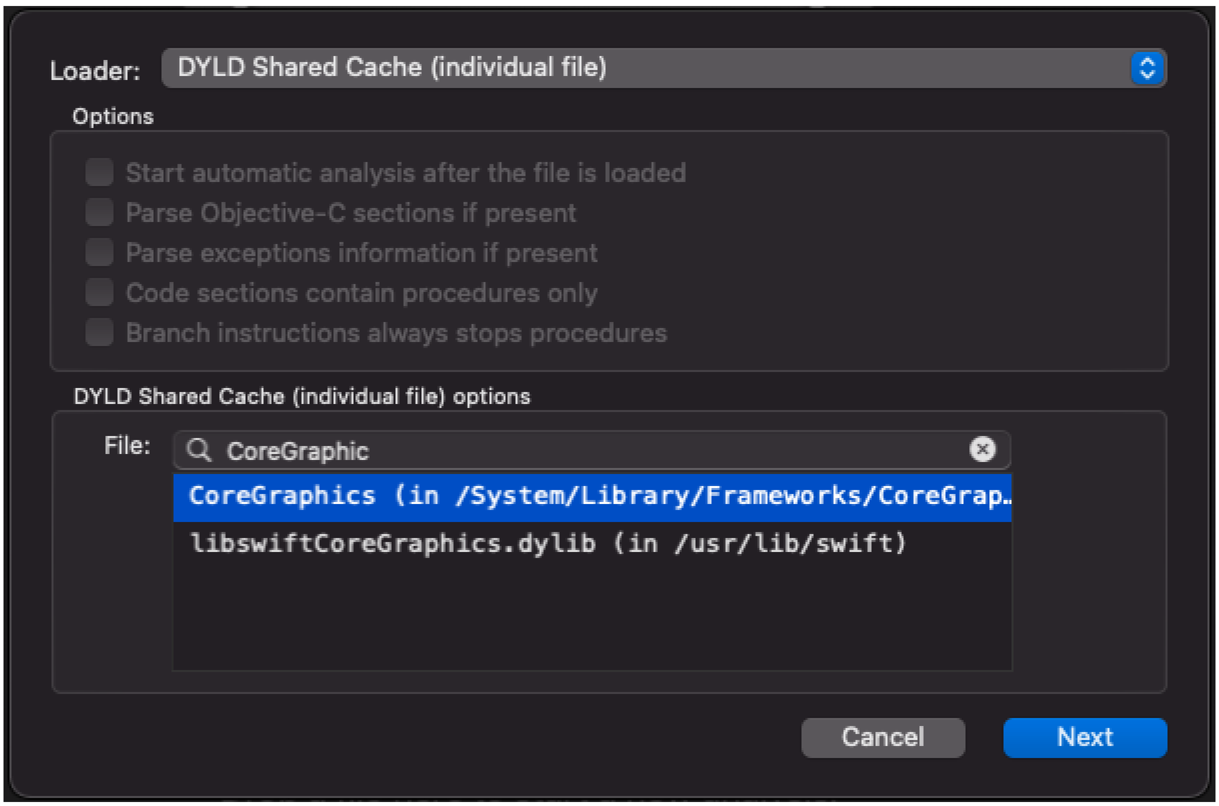
With the background out of the way, let’s get to reversing and find out what happened and how it was fixed!

In order to examine this, we first need to grab a vulnerable version of the `.dylib` (we will be using macOS 11.5.2) and a fixed version (macOS 11.6). I had Hopper and IDA for analysis as well, so for the sake of time, I utilized them both. First, we need to grab the

`CoreGraphics.dylib` from the two systems. At first, I was looking in the usual spot (`/System/Library/Frameworks/CoreGraphics`) and quickly noticed this was not the correct library. It turns out that on recent versions of macOS, many of the core frameworks are located in the dyld cache! This is a very large file, but is located in `/System/Library/dyld/dyld_shared_cache_x86_64`. (I'm using the x86_64 version).

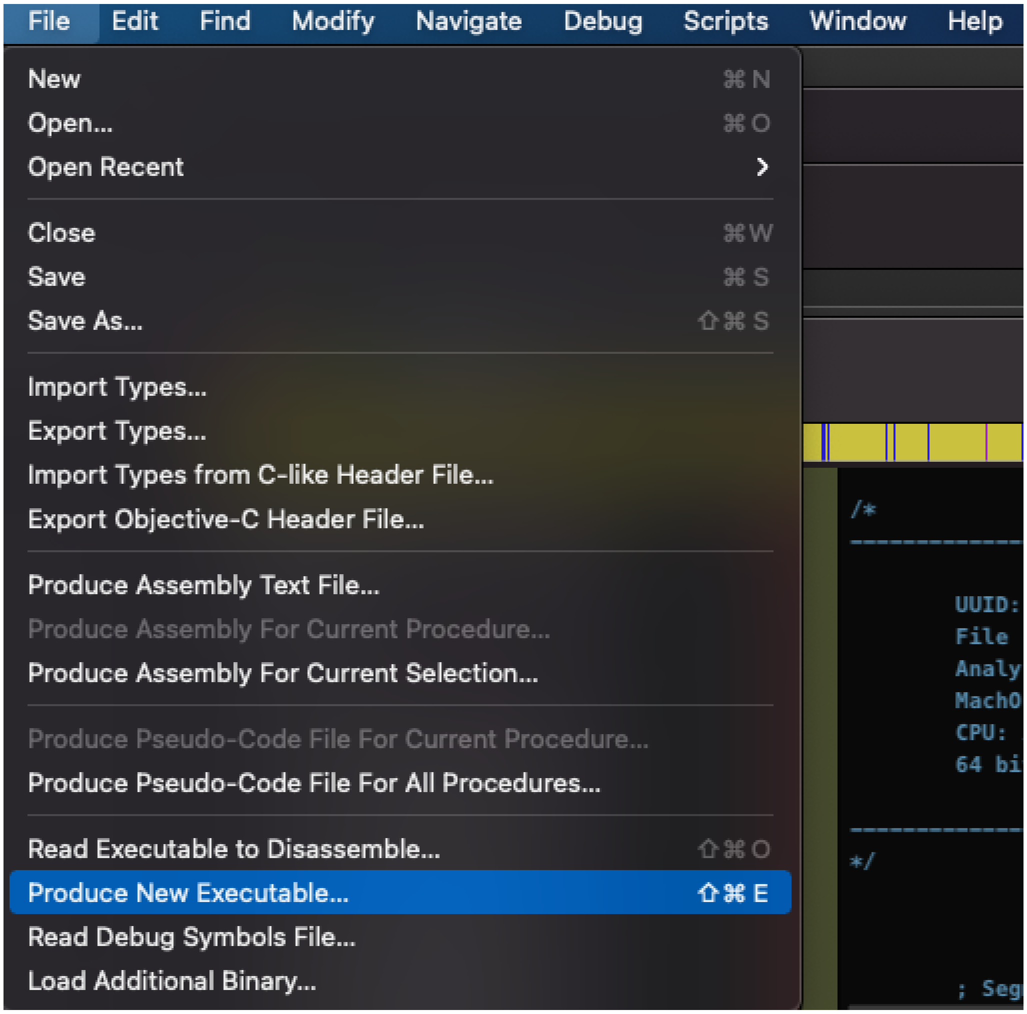
dyld cache from the respective folders

Armed with the knowledge of where the dyld cache is located, we need to extract the `CoreGraphics.dylib` from it. One of the simplest ways is to use the Hopper disassembler. Opening the `dyld_shared_cache_x86_64` file in Hopper presents you with myriad of Frameworks to examine. Of course, we will filter on the "CoreGraphics" one to open it up.



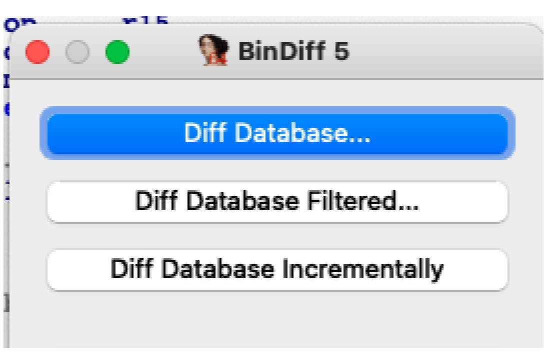
Hopper opening dyld_cache

From here, I was most interested in learning the differences between the 11.5.2 version and the 11.6 version. At this point, I decided to use Hopper to output the `CoreGraphics.dylib` to its own dedicated Mach-O file. To do this, we can use the "File->Produce New Executable" (or `cmd+shift+e`). Doing this for both the 11.5.2 `dyld_cache` and the 11.6 `dyld_cache` yields us the two `CoreGraphics.dylib` that we can easily analyze.



Hopper producing new executable

In order to diff these quickly, I decided to utilize IDA and BinDiff (we certainly can use other tools as well). So let's open both `CoreGraphics-11_5_2.dylib` and `CoreGraphics-11_6.dylib` in IDA, saving the corresponding `.i64` files. After closing both databases, I re-opened the `CoreGraphics-11_5_2.dylib` and launched BinDiff (`ctrl + 6`). After choosing "Diff Database…" and selecting the `CoreGraphics-11_6.i64` database, we wait for BinDiff to do its magic! It's not that bad actually. If you've not used BinDiff, the matching functions is quite useful. It also gives a guide for what has changed within a function. The BinDiff manual, from Zynamics site, gives a good description of the "Matched Functions Subview" and explains the "change" column.



BinDiff

Open BinDiff from the 11.5.2 version (primary) and use the `.i64` db for the 11.6 version (secondary)

We notice that there are 10 functions that have changed. It turns out that there was an API change or parameter size change to one of them (one of the parameters was removed), thus 4 of these functions aren't as "different" as they first appear. In Figure 8 below, the left most column is the similarity. 1.00 is identical* while lower values are less similar. We notice that there are a few entries with 0.99 similarity. These functions are mostly similar up to variance of some number of Instructions (I). The 0.92 similar function is the one of interest to us (some of the other functions are also worth examining…perhaps for another blogpost!). The "G", in the 3rd column (change) indicates there is a graph change (number of basic blocks or number of edges differ). There are also differences in the branch inversion, indicated by the "J". The "L" indicates the number of loops has changed. The graph structure is an important change to look at, as this could indicate a new branch condition was added or altered!

0.99	0.99	-I-----	00007FFF250F...	sub_00007FFF250F5730	00007FFF250F...	sub_00007FFF250F6754
0.99	0.99	-I-----	00007FFF250F...	sub_00007FFF250F524F	00007FFF250F...	sub_00007FFF250F5F9F
0.99	0.99	-I-----	00007FFF250F...	upsample_provider_get_bytes_at_position_inner	00007FFF250F...	upsample_provider_get_bytes_at_position
0.98	0.98	-I-----	00007FFF24E6...	sub_00007FFF24E69C82	00007FFF24E6...	sub_00007FFF24E6A962
0.92	0.98	GI-J-L-	00007FFF2524...	readTextRegionSeg_0	00007FFF25247...	readTextRegionSeg_0
0.89	0.99	GI-JE--	00007FFF24E6...	sub_00007FFF24E69396	00007FFF24E6...	sub_00007FFF24E69FD6
0.68	0.94	-I--E--	00007FFF250F...	call_to_API_change1	00007FFF250F...	sub_00007FFF250F62C9
0.65	0.90	-I--E--	00007FFF250F...	sub_00007FFF250F5549	00007FFF250F...	sub_00007FFF250F6291
0.63	0.68	GI-JEL-	00007FFF24EF1...	sub_00007FFF24EF19A4	00007FFF24EF2...	sub_00007FFF24EF2684
0.62	0.95	GI-----	00007FFF250F...	API_changed	00007FFF250F...	sub_00007FFF250F6301

Showing the differences to focus our analysis!

For this post, the most interesting function related to the `JBIG2` processing that differs between the 2 versions is located at: `00007FFF252466E0` (11.5.2 version) and `00007FFF25247710` (11.6 version) (In Figure 8, this is the `readTextRegionSeg_0` named routine).

This is the `JBIG2::readTextRegionSeg` function. As you can see, I didn't have symbols when doing this, however, I did notice some interesting strings present in the `CoreGraphics.dylib`, which turned out to be very useful in piecing together the code paths (obviously symbols would greatly help here, but even without them, we can identify the root cause…with a little help from open source software!)

Utilizing the strings located in the dylib, and the source code for a `JBIG2Stream processor`, we can match up some of the code!

Using source code as a guide, we can look at the issue in source and then match it to the disassembled version confirming the existence of the vulnerability in the 11.5.2 version.

As we can see below, the numSyms variable (an unsigned 32-bit integer), is incremented by the size of the currently processed segment. Thus, if there is more than one `jbig2SegSymbolDict` segment, `numSyms` will be updated with the size of that segment. This can lead to an integer overflow as there is no checking surrounding this area.

```
// get symbol dictionaries and tables
numSyms = 0;
for (i = 0; i < nRefSegs; ++i) {
  if ((seg = findSegment(refSegs[i]))) {
      if (seg->getType() == jbig2SegSymbolDict) {
              numSyms += ((JBIG2SymbolDict *)seg)->getSize();
       } else if (seg->getType() == jbig2SegCodeTable) {
              codeTables.push_back(seg);
       }
    } else {
      error(errSyntaxError, curStr->getPos(),
            "Invalid segment reference in JBIG2 text region");
      return;
    }
 }
```

As you can see from the disassembly below from the vulnerable version (11.5.2), the `add eax, [rbx+0ch]` (which is a 32-bit calculation), has no checking to ensure this hasn't wrapped. Thus, we have an integer overflow in which `numSyms` could wrap around.

```
__text:00007FFF25246A56 nRefSegs_loop:
__text:00007FFF25246A56                 mov     esi, [r13+r12*4+0]
__text:00007FFF25246A5B                 mov     rdi, r14
__text:00007FFF25246A5E                 call    findSegment
__text:00007FFF25246A63                 test    rax, rax
__text:00007FFF25246A66                 jz      loc_7FFF25246CDD
__text:00007FFF25246A6C                 mov     rbx, rax
__text:00007FFF25246A6F                 mov     rax, [rax]
__text:00007FFF25246A72                 mov     rdi, rbx
__text:00007FFF25246A75                 call    qword ptr [rax+10h]   ; getType()
__text:00007FFF25246A78                 cmp     eax, 1                ;
jbig2SegSymbolDict
__text:00007FFF25246A7B                 jnz     short loc_7FFF25246A8E
__text:00007FFF25246A7D                 mov     eax, dword ptr [rbp+numSyms]
__text:00007FFF25246A83                 add     eax, [rbx+0Ch]        ; numSyms +=
getSize()
__text:00007FFF25246A83                                               ; no overflow
check here!
__text:00007FFF25246A86                 mov     dword ptr [rbp+numSyms], eax
__text:00007FFF25246A8C                 jmp     short loc_7FFF25246AA7
__text:00007FFF25246A8E ; -----------------------------------------------------------
---
__text:00007FFF25246A8E
__text:00007FFF25246A8E loc_7FFF25246A8E:
__text:00007FFF25246A8E                 mov     rax, [rbx]
__text:00007FFF25246A91                 mov     rdi, rbx
__text:00007FFF25246A94                 call    qword ptr [rax+10h]   ; getType()
__text:00007FFF25246A97                 cmp     eax, 3                ;
jbig2SegCodeTable
__text:00007FFF25246A9A                 jnz     short loc_7FFF25246AA7
__text:00007FFF25246A9C                 mov     rdi, r15
__text:00007FFF25246A9F                 mov     rsi, rbx
__text:00007FFF25246AA2                 call    push_back
```

As we noted earlier, an integer overflow is often paired with 1 or more other mistakes. For example, it is used in an allocation routine to allocate dynamic memory. That is exactly the case here!

In the assembly below, we can see the `numSyms` being moved into `EDI` (prepping for the first argument to `gmallocn`). The `numSyms` value is controlled by the attacker. For example, we could have one segment be `0xFFFFFFFF` and the other be 2. We could also use `0x80000000` and `0x80000001`. The goal, of course, is to get `numSyms` to be a small number so the allocator, `gmallocn`, will create a small allocation.

```
__text:00007FFF25246AC1                 mov     edi, dword ptr [rbp+numSyms]
__text:00007FFF25246AC7                 cmp     edi, 2
__text:00007FFF25246ACA                 jb      short loc_7FFF25246ADB
__text:00007FFF25246ACC                 xor     ecx, ecx
__text:00007FFF25246ACE                 mov     eax, 1
__text:00007FFF25246AD3
__text:00007FFF25246AD3 loc_7FFF25246AD3:
__text:00007FFF25246AD3                 inc     ecx
__text:00007FFF25246AD5                 add     eax, eax
__text:00007FFF25246AD7                 cmp     eax, edi
__text:00007FFF25246AD9                 jb      short loc_7FFF25246AD3
__text:00007FFF25246ADB
__text:00007FFF25246ADB loc_7FFF25246ADB:
__text:00007FFF25246ADB                 mov     [rbp+var_2C4], ecx
__text:00007FFF25246AE1                 mov     esi, 8
__text:00007FFF25246AE6                 call    gmallocn
__text:00007FFF25246AEB                 mov     r8, rax
__text:00007FFF25246AEE                 xor     ebx, ebx
```

If we assume the `numSyms` was 1 following the overflow, `gmallocn` will allocate an 8-byte region for this. But where does this small allocation get used? And can we get more data to be copied into this buffer than was allocated?

Luckily we don't have far to go to see where there is an issue! First, we will look at the source code. We notice that this loop has similar processing to the vulnerable overflow one. In particular, it processes the `jbig2SegSymbolDict` segment. In this code path, we can see that the `getSize` method is called again and the bounds of the loop are tied to this. Since `getSize` returns an unsigned int (and `k` is already an unsigned int), this comparison is unsigned. Thus, even if `getSize` is `0x80000000`, this portion will execute.

As you can see on line 2004, the `syms` variable receives the bitmap. This `syms` was the result of the `gmallocn` allocation. Recall that only 8-bytes were allocated, in our example. But the `getSize` could be much larger, resulting in a heap buffer overflow!

```
    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); <-- overflow!
                }
            }
        }
    }
```

Let’s confirm the existence of this in the 11.5.2 code as well. From the code below, we can see that the `getBitmap_copyloop` is unbounded! Thus, a heap buffer overflow exists!

```
__text:00007FFF25246AF0
__text:00007FFF25246AF0 loc_7FFF25246AF0:
__text:00007FFF25246AF0                 mov     r15, r8
__text:00007FFF25246AF3                 mov     esi, [r13+rbx*4+0]
__text:00007FFF25246AF8                 mov     rdi, r14
__text:00007FFF25246AFB                 call    findSegment
__text:00007FFF25246B00                 test    rax, rax
__text:00007FFF25246B03                 jz      short loc_7FFF25246B53
__text:00007FFF25246B05                 mov     r12, rax
__text:00007FFF25246B08                 mov     rax, [rax]
__text:00007FFF25246B0B                 mov     rdi, r12
__text:00007FFF25246B0E                 call    qword ptr [rax+10h]    ; getType()
__text:00007FFF25246B11                 cmp     eax, 1                 ;
jbig2SegSymbolDict
__text:00007FFF25246B14                 jnz     short loc_7FFF25246B53
__text:00007FFF25246B16                 mov     eax, [r12+0Ch]         ; getSize()
__text:00007FFF25246B1B                 test    rax, rax
__text:00007FFF25246B1E                 mov     r8, r15
__text:00007FFF25246B21                 jz      short loc_7FFF25246B56
__text:00007FFF25246B23                 mov     r9, [rbp+var_2C0]
__text:00007FFF25246B2A                 mov     edx, r9d
__text:00007FFF25246B2D                 xor     ecx, ecx
__text:00007FFF25246B2F
__text:00007FFF25246B2F getBitmap_copyloop:
__text:00007FFF25246B2F                 lea     esi, [rdx+rcx]
__text:00007FFF25246B32                 mov     rdi, [r12+10h]
__text:00007FFF25246B37                 mov     rdi, [rdi+rcx*8]
__text:00007FFF25246B3B                 mov     [r8+rsi*8], rdi    ; leads to a heap
overflow
__text:00007FFF25246B3F                 inc     rcx
__text:00007FFF25246B42                 cmp     rax, rcx
__text:00007FFF25246B45                 jnz     short getBitmap_copyloop
```

Unfortunately, I did not have a sample to examine, so I could not confirm how the specific sample that CitizenLab had performed the attack.

In order to examine the fix, we need to look at the 11.6 version of the `CoreGraphics.dylib`. I would've expected to see an integer overflow check in the calculation of `numSyms` in the first loop. However, that is not the case. Below is the 11.6 version of the processing loop which is identical to 11.5.2! Maybe Apple will send out a proper fix soon :-)

```
__text:00007FFF25247A79 nRefSegs_loop:
__text:00007FFF25247A79                 mov     esi, [r12+r14*4]
__text:00007FFF25247A7D                 mov     rdi, r13
__text:00007FFF25247A80                 call    findSegment
__text:00007FFF25247A85                 test    rax, rax
__text:00007FFF25247A88                 jz      loc_7FFF25247D1D
__text:00007FFF25247A8E                 mov     rbx, rax
__text:00007FFF25247A91                 mov     rax, [rax]
__text:00007FFF25247A94                 mov     rdi, rbx
__text:00007FFF25247A97                 call    qword ptr [rax+10h]  ; getType()
__text:00007FFF25247A9A                 cmp     eax, 1               ;
jbig2SegSymbolDict
__text:00007FFF25247A9D                 jnz     short loc_7FFF25247AB0
__text:00007FFF25247A9F                 mov     eax, [rbp+numSyms]
__text:00007FFF25247AA5                 add     eax, [rbx+0Ch]       ; numSysm +=
getSize()
__text:00007FFF25247AA5                                              ; still no
overflow check!
__text:00007FFF25247AA5                                              ; even in
patched/11.6!
__text:00007FFF25247AA8                 mov     [rbp+numSyms], eax
__text:00007FFF25247AAE                 jmp     short loc_7FFF25247ACD
__text:00007FFF25247AB0 ; ----------------------------------------------------------
---
__text:00007FFF25247AB0
__text:00007FFF25247AB0 loc_7FFF25247AB0:
__text:00007FFF25247AB0                 mov     rax, [rbx]
__text:00007FFF25247AB3                 mov     rdi, rbx
__text:00007FFF25247AB6                 call    qword ptr [rax+10h]   ; getType()
__text:00007FFF25247AB9                 cmp     eax, 3                ;
jbig2SegSodeTable
__text:00007FFF25247ABC                 jnz     short loc_7FFF25247ACD
__text:00007FFF25247ABE                 mov     rdi, [rbp+var_2F0]
__text:00007FFF25247AC5                 mov     rsi, rbx
__text:00007FFF25247AC8                 call    push_back
```

Hrmm…not quite what I was expecting to see, but that's OK..there are other changes in this function. Recall that we noted that the integer overflow itself doesn't always lead to an issue, but it is usually paired with 1 or more other conditions. In this case, there are 2 other conditions that lead to the exploitable case. First, as we saw, the small `numSyms` value is used to allocate a memory region. With a small allocated buffer and the second issue of the copy loop using the larger values for its bounds (i.e. `getSize`), we have a recipe for the heap buffer overflow!

Based on that, and so far the fact that neither the `numSyms` calculation, nor the `gmallocn` area were changed, we can hope that this is fixed in the copy loop! And this is exactly what happened.

We can see below that we only go into the `getBitmap_copyloop` for the `numSyms` times. But this is only half of the problem. Since `getBitmap` is called in a loop, they also need to make sure that they stop the loop early there as well!

You can see that change in the `getBitmap_copyloop`, where they are now checking not only against the size of the segment (seen at `00007FFF25247B6B`), but they are also checking to ensure that the data copied to that point won't exceed the allocated buffer size (seen at `00007FFF25247B5B` and `00007FFF25247B7F`)

```
__text:00007FFF25247B23 loc_7FFF25247B23:
__text:00007FFF25247B23     mov     esi, [r12+r14*4]
__text:00007FFF25247B27     mov     rdi, r13
__text:00007FFF25247B2A     call    findSegment
__text:00007FFF25247B2F     test    rax, rax
__text:00007FFF25247B32     jz      short loc_7FFF25247B87
__text:00007FFF25247B34     mov     rbx, rax
__text:00007FFF25247B37     mov     rax, [rax]
__text:00007FFF25247B3A     mov     rdi, rbx
__text:00007FFF25247B3D     call    qword ptr [rax+10h]    ; getType()
__text:00007FFF25247B40     cmp     eax, 1                 ; jbig2SegSymbolDict
__text:00007FFF25247B43     jnz     short loc_7FFF25247B87
__text:00007FFF25247B45     cmp     r15d, [rbp+numSyms]    ; new check to make sure
we aren't
__text:00007FFF25247B45                                    ; going beyond the number
of symbols
__text:00007FFF25247B45                                    ;
__text:00007FFF25247B45                                    ; r15 is the 'counter' for
that
__text:00007FFF25247B45                                    ; originally set to 0
__text:00007FFF25247B4C     jnb     short loc_7FFF25247B87
__text:00007FFF25247B4E     mov     ecx, [rbx+0Ch]         ; effectively ecx =
getSize();
__text:00007FFF25247B51     mov     r15d, r15d
__text:00007FFF25247B54     mov     rdx, [rbp+orig_numSyms]
__text:00007FFF25247B5B     sub     rdx, r15               ; keep track of symbols
copied
__text:00007FFF25247B5E     mov     rax, [rbp+var_318]
__text:00007FFF25247B65     lea     rsi, [rax+r15*8]
__text:00007FFF25247B69     xor     eax, eax               ; copy loop counter
__text:00007FFF25247B6B
__text:00007FFF25247B6B getBitmap_copyloop:
__text:00007FFF25247B6B     cmp     rcx, rax               ; normal getSize() check
__text:00007FFF25247B6E     jz      short loc_7FFF25247B84
__text:00007FFF25247B70     mov     rdi, [rbx+10h]
__text:00007FFF25247B74     mov     rdi, [rdi+rax*8]
__text:00007FFF25247B78     mov     [rsi+rax*8], rdi
__text:00007FFF25247B7C     inc     rax
__text:00007FFF25247B7F     cmp     rdx, rax              ; this check ensures they
won't write
__text:00007FFF25247B7F                                   ;  out of bounds in the
copy loop!
__text:00007FFF25247B82     jnz     short getBitmap_copyloop
__text:00007FFF25247B84
__text:00007FFF25247B84 loc_7FFF25247B84:
__text:00007FFF25247B84     add     r15d, eax
__text:00007FFF25247B87
__text:00007FFF25247B87 loc_7FFF25247B87:
__text:00007FFF25247B87     inc     r14
__text:00007FFF25247B8A     cmp     r14, [rbp+nRefSegs]
__text:00007FFF25247B91     jnz     short loc_7FFF25247B23
```

This was certainly not the expected patch path when I first recognized the vulnerability. I would've thought the overflow would've been fixed at the point of calculation of numSyms. There may be a reason this is not the case. Perhaps that they still want the processing to occur even in the case of some 'malformed' PDFs for whatever reason. Who knows!

`readSymbolDictSeg` and Differences in the Patch

As we alluded to earlier, another method has a very similar processing loop, but it was actually protected from the integer overflow before this release! In fact, the fix in this code checks for the integer overflow when calculating the number of symbols!

Using our `JBIG2` source code as an example, we can see the following processing. On lines 1536-1539, we see the integer overflow check to ensure that when the statement on line 1540 is executed, it won't overflow!

In addition, they are checking to ensure the number of new symbols hasn't exceeded the bounds (lines 1548-1549)

```
1527     // get referenced segments: input symbol dictionaries and code tables
1528     numInputSyms = 0;
1529     for (i = 0; i < nRefSegs; ++i) {
1530         // This is need by bug 12014, returning false makes it not crash
1531         // but we end up with a empty page while acroread is able to render
1532         // part of it
1533         if ((seg = findSegment(refSegs[i]))) {
1534             if (seg->getType() == jbig2SegSymbolDict) {
1535                 j = ((JBIG2SymbolDict *)seg)->getSize();
1536                 if (numInputSyms > UINT_MAX - j) {
1537                     error(errSyntaxError, curStr->getPos(),
                               "Too many input symbols in JBIG2 symbol dictionary");
1538                     goto eofError;
1539                 }
1540                 numInputSyms += j;
1541             } else if (seg->getType() == jbig2SegCodeTable) {
1542                 codeTables.push_back(seg);
1543             }
1544         } else {
1545             return false;
1546         }
1547     }
1548     if (numInputSyms > UINT_MAX - numNewSyms) {
1549         error(errSyntaxError, curStr->getPos(),
                   "Too many input symbols in JBIG2 symbol dictionary");
1550         goto eofError;
1551     }
```

In the assembly from 11.5.2, we can see the overflow check at addresses `00007FFF2524576D` - `00007FFF25245774`, with the branch at `00007FFF25245774` going down the error path:

```
__text:00007FFF25245748 loc_7FFF25245748:
__text:00007FFF25245748                 mov     rax, [rbp+var_68]
__text:00007FFF2524574C                 mov     esi, [rax+rbx*4]
__text:00007FFF2524574F                 mov     rdi, r14
__text:00007FFF25245752                 call    findSegment
__text:00007FFF25245757                 test    rax, rax
__text:00007FFF2524575A                 jz      short loc_7FFF25245791
__text:00007FFF2524575C                 mov     r12, rax
__text:00007FFF2524575F                 mov     rax, [rax]
__text:00007FFF25245762                 mov     rdi, r12
__text:00007FFF25245765                 call    qword ptr [rax+10h]    ; getType()
__text:00007FFF25245768                 cmp     eax, 1                 ;
jbig2SegSymbolDict
__text:00007FFF2524576B                 jnz     short loc_7FFF25245776
__text:00007FFF2524576D                 add     r13d, [r12+0Ch]
__text:00007FFF25245772                 jnb     short loc_7FFF25245791 ; integer
overflow check
__text:00007FFF25245774                 jmp     short integer_overflow
```

As you can see, this overflow check was done during the calculation of the number of symbols. This is due to the `jnb` instruction. The add instruction will perform both signed and unsigned operation and adjust the `Overflow Flag` (`OF`) and/or `Carry Flag` (`CF`) for signed and unsigned respectively. The `jnb` instruction (a pseudonym for `jnc`) indicates to jump if the carry flag is 0 (i.e. no integer wrapping occurred). In this case, this is the 'good' path, whereas if the `CF` was set, this would indicate an integer wrapping and the corresponding error path is taken!

On the other hand, the `readTextRegionSeg` method, the `numSyms` can still overflow, however, in the processing loop when the `getBitmp` method is copying to the allocated region, there is a check to ensure that this data is not overflowed.

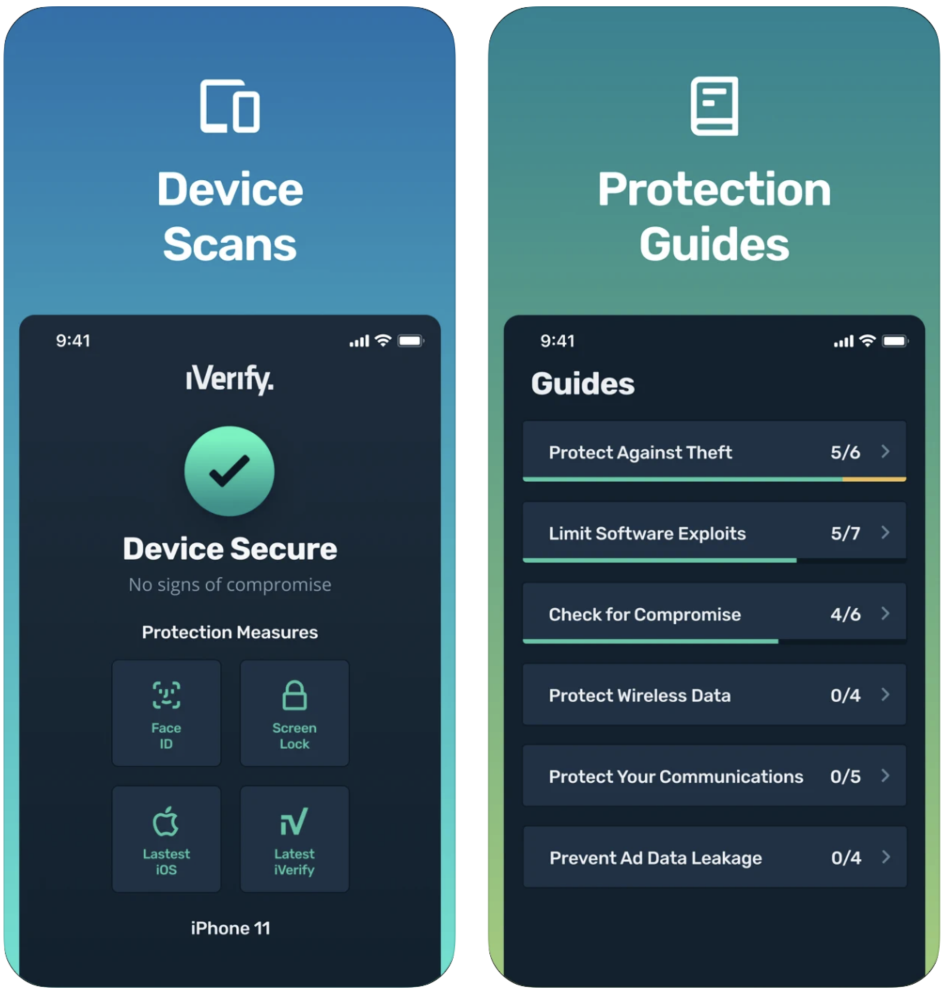
Based on the analysis and the abundance of common strings, it seems that Apple is likely using an opensource version of the `JBIG2` processing, and making their own modifications. (Admittedly, I did search for their notes on this, but didn't find it…if anyone confirms that they are using that would be awesome). It does seem that a different developer implemented the fix in `CVE-2021-30860` than the one found in the `readSymbolDictSeg` method.

Concluding Thoughts

There were other functions that were patched as well. For example, in the 11.6 version, it is worth analyzing the functions at address `00007FFF24EF2684` and `00007FFF250F6301` . Perhaps for another blog post…

As we noted, this vulnerability is (well prior to the patch) exploitable through a crafted iMessage without any user-interaction. In other words, a specially crafted PDF file could be sent to an iMessage recipient, and the victim's `IMTranscoderAgent` begins processing the malicious payload outside of the BlastDoor sandbox. As noted in the beginning of this post, hopefully Apple will also update BlastDoor and prevent these dangerous file formats from being processed outside the Sandbox environment!

Apple’s iDevices have gotten more secure especially from allowing their system to be modified upon reboot. Thus, a good practice for iOS users is to a) update when updates are available and b) reboot the phone every so often! Of course this won’t stop these 0day attacks, but it is at least a good security practice. It would be worth downloading iVerify to help test for common infections as well as for recommendations to increase the security posture of your device!



iVerify

Part 0x2

…stay tuned! 🧐

This website uses cookies to improve your experience.