

# Analyzing The ForcedEntry Zero-Click iPhone Exploit Used By Pegasus

By Mickey Jin Sep 15, 2021 Read time: 4 min (1035 words)

Published: 2021-09-15 · Archived: 2026-04-05 18:40:01 UTC

## Exploits & Vulnerabilities

Citizen Lab has released a report on a new iPhone threat dubbed ForcedEntry. This zero-click exploit seems to be able to circumvent Apple's BlastDoor security, and allow attackers access to a device without user interaction.

---

Citizen Lab has released a [report open on a new tab](#) detailing sophisticated iPhone exploits being used against nine Bahraini activists. The activists were reportedly hacked with the NSO Group's Pegasus spyware using two zero-click iMessage exploits: [Kismet open on a new tab](#), which was identified in 2020; and [ForcedEntry open on a new tab](#), a new vulnerability that was identified in 2021. Zero-click attacks are labeled as sophisticated threats because unlike typical malware, they do not require user interaction to infect a device. The latter zero-click spyware is particularly notable because it can bypass security protections such as BlastDoor, which was designed by Apple to protect users against zero-click intrusions such as these.

According to Citizen Lab's report, Kismet was used from July to September 2020 and was launched against devices running at least iOS 13.5.1 and 13.7. It was likely not effective against the iOS 14 update in September. Then, in February 2021, the NSO Group started deploying the zero-click exploit that managed to circumvent BlastDoor, which Citizen Lab calls ForcedEntry. Amnesty Tech, a global collective of digital rights advocates and security researchers, also observed zero-click iMessage exploit activity during this period and referred to it as [Megalodon open on a new tab](#).

## Diving into ForcedEntry

According to the report from Citizen Lab, when the ForcedEntry exploit was launched against the victim's device, the device logs showed two types of crashes. The first crash apparently happened when invoking ImageIO's functionality for rendering Adobe Photoshop PSD data.

Our analysis focuses on the second crash, which is detailed in Figure 1. This crash happened when invoking CoreGraphics' functionality for decoding JBIG2-encoded data in a PDF file. This analysis is solely based on samples from Citizen Lab; no new samples were obtained.

```
Thread 2 name: Dispatch queue: IMTranscoderNormalPriorityQueue
Thread 2 crashed:
0: CoreGraphics 0x181d6e228 __ZN11JBIG2Stream17readTextRegionSegEjjjPjj + 980
1: CoreGraphics 0x181d6e20c __ZN11JBIG2Stream17readTextRegionSegEjjjPjj + 872
2: CoreGraphics 0x181d6c7c __ZN11JBIG2Stream12readSegmentsEv + 1988
3: CoreGraphics 0x181d6be70 __ZN11JBIG2Stream5resetEv + 260
4: CoreGraphics 0x181d79f9c __ZL10read_bytesPvS_m + 1024
5: CoreGraphics 0x181d1e324 __jbig2_filter_refill + 128
6: CoreGraphics 0x181d8d998 __CGPDFSourceRefill + 196
7: CoreGraphics 0x181d8cfa4 __CGPDFSourceGete + 36
8: CoreGraphics 0x181d63088 __xref_stream_read_section + 188
9: CoreGraphics 0x181d62e60 __xref_stream_create + 828
10: CoreGraphics 0x181d62a56 __CGPDFStreamCreate + 112
11: CoreGraphics 0x181d26694 __pdf_xref_create + 1748
12: CoreGraphics 0x181d06eb0 __CGPDFDocumentCreateWithProvider + 280
13: ImageIO 0x181d0fd44 __Z19CreateSessionPDFRefP10IIOScannerPb + 112
14: ImageIO 0x181d92404 __ZN14IIIO_Reader_PDF22updateSourcePropertiesEP19IIIOImageReadSessionP13IIODictionaryS3_S3_P19CGImageSourceStatus + 84
15: ImageIO 0x181d1387c __ZN14IIIOImageSource13getPropertiesEP13IIODictionary + 408
16: ImageIO 0x181d13944 __ZN14IIIOImageSource14copyPropertiesEP13IIODictionary + 16
17: ImageIO 0x181d17f00 __CGImageSourceCopyProperties + 244
18: IMSharedUtilities 0x181d7b974 readFileProperties:fromImageSource:error: + 48
19: IMSharedUtilities 0x181d7c740 readFileProperties:fromImageSource:withUpdatedLoopCount:error: + 84
20: IMSharedUtilities 0x181d7cd34 copyGifFromPath:toDestinationPath:error: + 264
21: IMTranscoderAgent 0x181d258c8
```

Figure 1. This image from Citizen Lab shows a Symbolicated Type Two crash for ForcedEntry on an iPhone 12 Pro Max running iOS 14.6. The red highlights from Trend Micro Research.

From this crash log, we can deduce three interesting points: First, the zero-click attack is dependent on iMessage attachment parsing. Next, the slide of dyld\_shared\_cache is 0, which means all the system modules are loaded into a fixed address. Lastly, the crash point 0x181d6e228 is not the first place of vulnerability exploitation. We discuss the details of these conclusions in the following sections.

### Root cause of CVE-2021-30860

The vulnerability is inside the function **JBIG2Stream::readTextRegionSeg** of CoreGraphics.framework. The crash point **0x181d6e228** (as seen in box 3 in the preceding figure) is at line 161 of the function **JBIG2Stream::readTextRegionSeg** of the following screenshot:

```
114 numSyms = 0;
115 nRefSegs_1 = nRefSegs;
116 refSegs_1 = (int *)refSegs;
117 v28 = nRefSegs;
118 do
119 {
120 Segment = (JBIG2SymbolDict *)JBIG2Stream::findSegment(this, *refSegs_1);
121 if ( !Segment )
122 {
123 v47 = (*(__int64 (__fastcall **))(JBIG2Stream *))(this + 40LL)(this);
124 error(v47, "Invalid segment reference in JBIG2 text region");
125 j__free(*(void **)v106);
126 operator delete(v106);
127 return;
128 }
129 v30 = Segment;
130 if ( Segment->vfptr->getType(Segment) == jbig2SegSymbolDict )
131 {
132 numSyms += v30->size;
133 }
134 else if ( v30->vfptr->getType(v30) == jbig2SegCodeTable )
135 {
136 GList::append(v106, v30);
137 }
138 ++refSegs_1;
139 --v28;
140 }
141 while ( v28 );
142 v89 = v12;
143 v91 = v14;
144 v31 = 0;
145 if...
146 syms = (__QWORD *)gmallocn(numSyms, 8u);
147 i_1 = 0LL;
148 k = 0LL;
149 do
150 {
151 seg = (JBIG2SymbolDict *)JBIG2Stream::findSegment(this, refSegs[i_1]);
152 if ( seg
153 && (symbolDict = seg, seg->vfptr->getType(seg) == jbig2SegSymbolDict)
154 && (size = symbolDict->size, (_DWORD)size) )
155 {
156 bitmaps = symbolDict->bitmaps;
157 do
158 {
159 v40 = (__int64)*bitmaps++;
160 kk = (unsigned int)(k + 1);
161 syms[(unsigned int)k] = v40; // crash here !!!
162 LODWORD(k) = k + 1;
163 --size;
164 }
165 while ( size );
166 }
167 else
168 {
169 kk = k;
170 }
171 ++i_1;
172 k = kk;
173 }
174 while ( i_1 != nRefSegs_1 );
```

00085228 \_\_ZN11JBIG2Stream17readTextRegionSegEjiiPjj:161 (181D6E228)

Figure 2. Screenshot of the function JBIG2Stream::readTextRegionSeg showing the crash point

First, it calculates the *numSyms* according to the JBIG2SymbolDict segment:

```

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->append(seg);
        }
    } else {
        error(getPos(), "Invalid segment reference in JBIG2 text region")
    }
}
delete codeTables;
return;
}
}

```

The type of `numSyms` is unsigned int, and the return type of function `seg->getSize()` is also unsigned int. Therefore, `numSyms` could be smaller than the size of one JBIG2Segment due to integer overflow. One example is `numSyms=1=(0x80000000+0x80000001) < 0x80000000`.

Then, it allocates the heap buffer `syms`, with the size `numSyms * 8` :

```

syms = (JBIG2Bitmap **)gmallocn(numSyms, sizeof(JBIG2Bitmap *));

```

Finally, it fills the `syms` with the value from bitmap:

```

kk = 0;
for (i = 0; i < nRefSegs; ++i) {
    if ((seg = findSegment(refSegs[i]))) {
        if (seg->getType() == jbig2SegSymbolDict) { //seg->getType() i
s a virtual function
            symbolDict = (JBIG2SymbolDict *)seg;
            for (k = 0; k < symbolDict->getSize(); ++k) {
                syms[kk++] = symbolDict->getBitmap(k); // crashed h
ere
            }
        }
    }
}
}
}

```

The loop times are dependent on the JBIG2Segment size, which could be larger than the buffer `syms` size. This leads to the out-of-bounds write access for the heap buffer `syms`.

### Looking at Apple's fix

Apple patched the function in iOS 14.8:

```
149 syms = (_QWORD *)gmallocn(numSyms, 8);
150 i_1 = 0LL;
151 kk = 0;
152 do
153 {
154     seg = (JBIG2SymbolDict *)JBIG2Stream::findSegment(this, refSegs[i_1]);
155     if ( seg )
156     {
157         symbolDict = seg;
158         v37 = seg->vfptr->getType(seg) != jbig2SegSymbolDict || kk >= numSyms;
159         if ( !v37 )
160         {
161             k = 0LL;
162             size = symbolDict->size;
163             do
164             {
165                 if ( size == k )
166                     break;
167                 syms[kk + k] = symbolDict->bitmaps[k];
168                 ++k;
169             }
170             while ( numSyms - (unsigned __int64)kk != k );
171             kk += k;
172         }
173     }
174     ++i_1;
175 }
176 while ( i_1 != nRefSegs );
177 v40 = syms;
178 v12 = v86;
```

000850AC \_\_ZN11JBIG2Streaml7readTextRegionSegEjiiPjj:158 (181D710AC)

Figure 3. Screenshot of the same function JBIG2Stream::readTextRegionSeg with fixes in place

We can see that Apple adds two new boundary checks (the red box in Figure 3), to avoid overflowing the syms buffer.

### On the Pegasus spyware exploitation

#### Disabling ASLR

The `dyld_shared_cache` of version iOS 14.6 (18F72) was loaded into IDA Pro for static analysis, after which a surprising result emerged. We were able to go to the addresses on the call stack directly without rebasing the segment.

As deduced from the screenshot in Figure 1 (see box 2), the slide of `dyld_shared_cache` is **0**. However, in common crash scenarios, these addresses should be in **slide**.

If the screenshot of the original crash log has not been modified, then the conclusion is worrying. It should be noted that Pegasus already disabled Address Space Layout Randomization (ASLR) before its exploitation.

#### Bypassing PAC

By inspecting the address `0x181d6e20c` from Frame 1 of the call stack trace, we can see that register x0, the return value of function `JBIG2Stream::findSegment`, is a subclass of `JBIG2Segment`:

```

CoreGraphics: text:0000000181D6E1E8      LDR      W1, [X26,X22,LSL#2] ; unsigned int
CoreGraphics: text:0000000181D6E1EC      MOV      X0, X19 ; this
CoreGraphics: text:0000000181D6E1F0      BL      ZN11JBIG2StreamIifindSegmentEj ; JBIG2St
CoreGraphics: text:0000000181D6E1F4      CBZ     X0, loc_181D6E23C
CoreGraphics: text:0000000181D6E1F8      MOV      X23, X0
CoreGraphics: text:0000000181D6E1FC      LDR      X8, [X0]
CoreGraphics: text:0000000181D6E200      LDRAA   X9, [X8,#0x10]!
CoreGraphics: text:0000000181D6E204      MOVK    X8, #0xFA4,LSL#48
CoreGraphics: text:0000000181D6E208      BIKAAA X9, X8 ; call virtual function getType()
CoreGraphics: text:0000000181D6E20C      CMP     W0, #1
CoreGraphics: text:0000000181D6E210      B.NE   loc_181D6E23C
CoreGraphics: text:0000000181D6E214      LDR     W8, [X23,#0xC]
00085208 0000000181D6E208: JBIG2Stream::readTextRegionSeg(uint,int,int,uint,uint *,uint)+364 (Synchron

```

```

class JBIG2Segment {
public:
...
    virtual JBIG2SegmentType getType() = 0;
private:
    uint segNum;
};

```

There are four kinds of subclasses that override the `getType()` virtual function, but the following code shows that they just return one of the enumerate values:

```

enum JBIG2SegmentType {
    jbig2SegBitmap,
    jbig2SegSymbolDict,
    jbig2SegPatternDict,
    jbig2SegCodeTable
};

```

For example, `JBIG2SymbolDict::getType` just returns `jbig2SegSymbolDict=1`:

```

0181D6B984 ; __int64__fastcall JBIG2SymbolDict::getType(JBIG2SymbolDict * __
0181D6B984 __ZN15JBIG2SymbolDict7getTypeEv ; DATA XREF: CoreGraphics
0181D6B984 MOV      W0, #jbig2SegSymbolDict
0181D6B988 RET
0181D6B988 ; End of function JBIG2SymbolDict::getType(void)

```

Therefore, the **frame 1** should have called the virtual function `seg->getType()`. But in actuality, it was already subverted to the current function itself (**frame 0**).

This shows that the virtual functions table of the object `JBIG2Segment` had already been replaced, and the pointer authentication code (PAC) security feature was bypassed. This is significant because the PAC security mechanism was developed to [help prevent zero-click hackingopen on a new tab](#). This also shows that the crash point is not the first place of the vulnerability exploitation.

### Conclusion and recommendations

From the view of attack technologies used, we can see that Pegasus is quite an advanced threat for iOS users. However, it seems that these attacks are being launched on very specific targets, rather than common users.

The information from the recent Pegasus attack is from the forensic analysis of Citizen Lab and Amnesty Tech, and we have not found Pegasus attack samples that are at large yet. We are actively searching and monitoring for these threats and will continue to share more details as our investigation continues.

Essentially, this attack is a very common file format parsing vulnerability. We previously discovered [CVE-2020-9883open on a new tab](#), a vulnerability similar to ForcedEntry, which could be exploited to do the same as what Pegasus has done here. ForcedEntry's key point is the exploit technology as it is still unknown how it is able to bypass the PAC and disable ASLR.

In the meantime, we strongly recommend [updating your device to iOS 14.8open on a new tab](#). As stated previously, common iOS users are not the target for attacks using this spyware. However, there are simple security steps that users can take. For example, concerned users can block iMessages from unknown senders, while a more drastic step would be to disable the iMessage function completely in the device's Preferences.

Tags

---

Source: [https://www.trendmicro.com/en\\_us/research/21/i/analyzing-pegasus-spywares-zero-click-iphone-exploit-forcedentry.html](https://www.trendmicro.com/en_us/research/21/i/analyzing-pegasus-spywares-zero-click-iphone-exploit-forcedentry.html)