An Investigation Into Pegasus's Attack of iOS Vulnerability CVE 2021–30860

Meenakshi Iyer

December 15, 2023

Table of Contents

List of Visuals iii
History of Pegasus.. 1
ForcedEntry, an iMessage Exploit 3
JBIG2 Vulnerability 4
Weird Machines 8
Apple's Fix 9
Chrysaor 10
Conclusion 10
Works Cited 11

List of Visuals

Image 1: Pegasus by the Numbers.. 3
Figure 2: Glyph Susbtitution.. 4
Image 3: ReadTextRegionSeg 5
Figure 4: Buffer Overflow 6
Figure 5: JBIG2 Bitmap 7
Figure 6: iOS Patch 9

History of Pegasus

Like most others in the early 2000s, Omri Lavie and Shavel Hulio were interested in entering the startup industry. The pair developed a tool that let IT representatives access entire control of the company's customers' devices remotely- *requiring that the user grant them permission first*. CommuniTake, their product and company, was launched in 2009. As their product gained traction, European intelligence companies developed interest in the product, and soon the idea to develop a tool that *didn't* rely on user authorization was born. In early 2010, NSO (Niv, Shavel, and Omri) Group was founded by Lavie, Hulio, and Niv Karmi (a former Mossad Intelligence Operative), and started employing dozens of military intelligence workers to help develop Pegasus. NSO Group took off, and compared to international competitors stuck in a time of run-of-the-mill email attachment malware, had more modern technologies to offer: SMS Infections. With Mexico being one of the first major customers, the money came pouring in, and the list of journalists, politicians, and drug traffickers targeted for surveillance increased.

NSO Group, as opposed to other malware companies, decided to stick exclusively to smartphones. As of 2020, more than two-thirds of their employees worked in research; specifically, they were scouring softwares for zero-day vulnerabilities, in which newly deployed code had vulnerabilities of which the public was not yet aware.

By taking advantage of these vulnerabilities, the group was able to extend the length of time for which their attacks are not defended. Moreover, Pegasus employed zero-click attacks, in which the user does not have to interact with the device to allow the software to corrupt it, making Pegasus go from discrete to practically invisible. The power in the attacker's hand posed a serious threat—to educate individuals on internet safety, and to beware scams is one level, and telling individuals there is nothing they can do about being attacked is another. Shavel Hulio himself is said to only use Samsung phones running Android, and he often updates them.

According to their website, NSO Group "creates technology that helps government agencies prevent and investigate terrorism and crime to save thousands of lives around the globe. " (NSO Group) However, the operations into which they market their software are highly secretive. They claim to work with governments to prevent terrorist attacks, break up pedophilia, sex, and drug trafficking rings, locate kidnapped children, etc. As of 2022, little was known about their client screening process, but according to an official statement on their website, "our vetting process goes beyond legal and regulatory requirements to ensure the lawful use of our technology as designed."

The image below quantifies that statement further. Notably, possibly six countries using the spyware had a history of abusing spyware to target civil society.

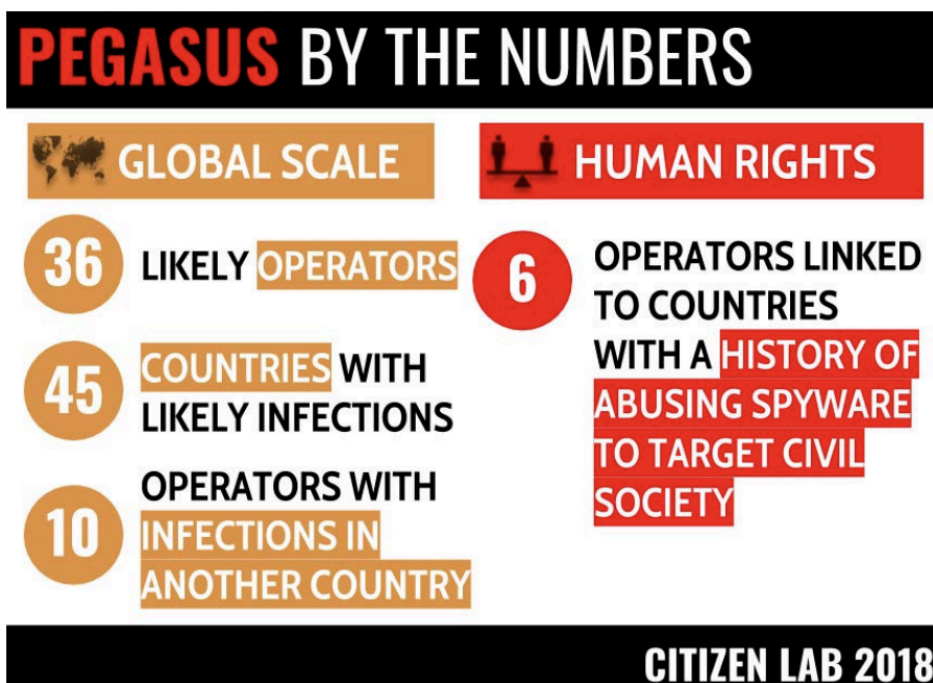

Image 1: PEGASUS BY THE NUMBERS

For example, the West African country of Togo (interestingly, a strong poltical ally with Israel) has long used torture and excessive force against peaceful opposition. In order to spy on peaceful opposition, the operator of Pegasus would deploy decoy websites with keywords such as "'*nouveau president*' (new president) and '*politiques info*' (political information)" (Marczak, Scott-Railton, McKune, Razzak, Deibert 10), to attract potential targets to the website and infect their devices with the spyware. There are several of these operators working in Israel, and other countries, including the Netherlands, Palestine, and Qatar.

What makes Pegasus stand out amongst other spyware is its elusiveness. Despite numerous attempts at educating people about the dangers of clicking scam links, or general internet safety, there is little that could be done to defend against Pegasus. Most importantly, individuals are unable to realize that malware was on their device due to its zero-click exploitation, meaning the user would not mistakenly click on a link, or navigate to a "shady" website.

Generally, spyware programs often exhibit telltale signs of their presence, such as increased CPU cycles and abnormal memory consumption. The collected data is then transmitted to the operator's servers through HTTP connections. As a result, operators would need to maintain domain names, which allow the malware to be tracked. One technique is fingerprinting, in which cryptographic hash values are generated and hidden within an executable to provide authentication. Another technique, DNS cache probing, is where the Domain Name System activity on a domain is monitored, and suspicious activity is investigated. These techniques offer

some insight into Pegasus's inner workings, but only to a limited extent. (Marczak, Scott-Railton, McKune, Razzak, Deibert 8)

Dubbed as military grade spyware, Pegasus is also capable of hacking through encryptions on most cell phones, converting them into an audio and visual tracking device, completely unbeknownst to the user. It took advantage of iOS and Android vulnerabilities, occurring in a multitude of places, such as Apple Music, iMessage, Photos App, and Homekit. Once the exploit is executed, Pegasus can use fake authentication in order to get kernel level access, and track location, all user communications (texts, phone calls, etc.), and much more, all this again without the knowledge or consent of the user.

In comparison to other zero-click spywares, Pegasus has not been deployed by cybercriminals for financial gain, but is instead associated with countries and governments, which raises questions about human rights, abuses of power, and suppression of dissent. Ethics behind this have long been discussed to no avail. One one hand, spyware used for security purposes can be powerful, and can contribute to all the goals that NSO Group prioritizes. On the other hand, when malware ends up in the wrong hands, it can be used in dangerous ways that harm civilians, and work against the goals of the organization. Since digital espionage is an important part of national security, lines defining how much privacy an individual is entitled to, especially when they are suspected of causing danger or terror, are constantly changing.

ForcedEntry, An iMessage Exploit

ForcedEntry, a new vulnerability identified in 2021, offers valuable insight into the inner workings of one of Pegasus's exploits. These zero-click attaches are sophisticated in how they infect a user's device, making devices running iOS easy targets. This exploit was an iMessage exploit and based on memory corruption. Such exploits are typically required to have the following: a memory corruption vulnerability, a way to break Address Space Layout Randomization (ASLR), a way to turn the vulnerability into remote code execution, and finally, a way to break out of a sandbox. The exploit was marked as Common Vulnerability Exposure (CVE) 2021–30860, in which PDF files disguised as GIF files injected JBIG2-encoded data to provoke an integer overflow.

The way Apple renders GIFs in iMessage is by repeatedly looping them, rather than having them only play once. In between when the message is received and when it is rendered on the user's screen, any file with the extension ".gif" is sent to a function that repeatedly uses the CoreGraphics API to render the source image into a new GIF file at the destination path. This means that any file with extension ".gif" (even if it is not actually a GIF file) is rendered into a

new GIF file. In this process, the ImageIO library is used to parse the source file using its image format, and not the file extension. For example, a file ending in ".gif" would be sent to be repeatedly rendered, and each time, call a codec associated with its actual file type. Therefore, a file that may not be a .gif, but has a .gif extension, could gain access to many image codecs, and increase the surface area of exploitation. Using this trick, NSO was able to target a vulnerability in the CoreGraphics PDF parser.

JBIG2 Vulnerability

In the CoreGraphics PDF parser, the JBIG2 standard (a domain specific image codec to tremendously increase compression rates) has a "partially lossy" method of compressing. The JBIG2 standard is an encoding/decoding method for black and white printed matter. The partially lossy method involves storing the bit difference between a substituted hieroglyphic and original hieroglyphic. The glyph character in this case is a letter. The image below depicts the difference.

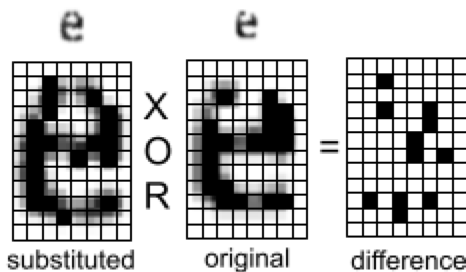

Figure 2: GLYPH SUBSTITUTION

This difference can be computed using the logical operator XOR. Rather than do the iteration in one go, it is done step by step using many logical operators to map these bits. Each step brings the rendered output closer to the original, which allows control over the level of "lossiness".

The format of the **JBIG2** class is a series of segments, which can be considered as a series of commands. The **JBIG2Bitmap**, a subclass of **JBIG2**, represents a rectangular array of pixels. A **JBIG2SymbolDict** class groups **JBIG2Bitmaps** together.

In the function **readTextRegionSeg** below, an unsigned integer **numSyms** is calculated according to the size of the **JBIG2SymbolDict**.

```
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)
      }
    }
  }
}
```

Image 3: **readTextRegionSeg**

numSyms is an unsigned integer (1) that is repeatedly incremented (2).

When two unsigned integers that are really large are added, it is possible that the result be smaller than the actual sum. For example, consider a number in binary that can only be represented in 4 bits. If we take the number 0100 and 1101, which are 4 and 13 respectively, we know the sum must be 10001, or 17 in decimal. The system can only represent numbers with 4 bits however, so it will save the last 4 bits, which make the number 0001, or 1. Thus, 4+13=1 because of integer overflow.

With careful calculations, it is possible to make **numSyms** overflow to a smaller value.

Then, that undersized value is used to allocate the size of the heap buffer, which **syms** points to (3). The heap buffer is a buffer that exists in the heap portion of memory, where all global variables and global data exist. With the loop (3), the **JBIG2Bitmap** pointer values are written into the undersized **syms**.

However, once the limit of the buffer is reached, the values would keep being written, to values out of the bounds of the buffer, leading to an out of bounds error. Commonly, out of bounds errors result in a crash that would stop the process immediately. However, there exists a nifty trick where the heap buffer is groomed so that the first few write offs of the end of this buffer corrupt the **GLIST** backing buffer, which stores all known **JBIG2** segments. With this overflow, the **GList** backing buffer is overwritten with **JBIG2Bitmaps** instead of **JBIG2Segments**. The image below depicts this process.

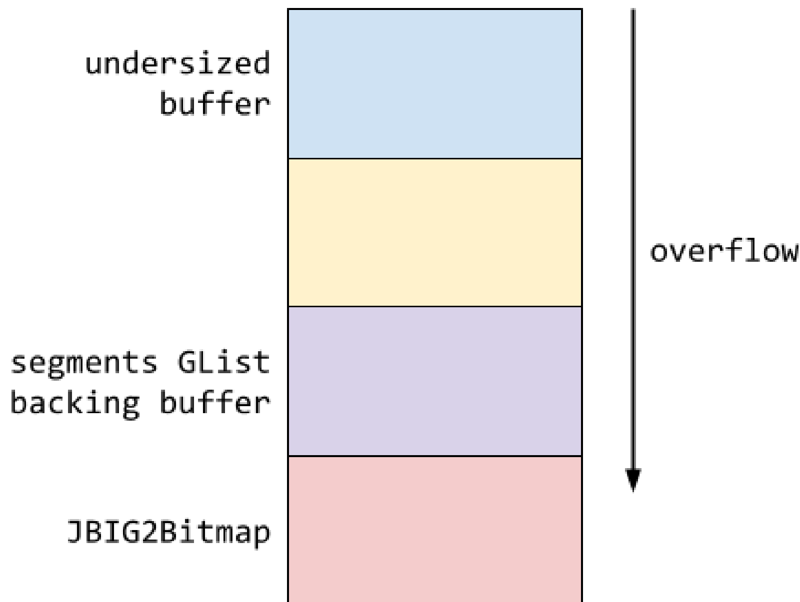

Figure 4: BUFFER OVERFLOW

Then, the attacker grooms the **JBIG2Bitmap** object that represents the current page. These bitmaps are wrappers around a backing buffer that store its width and height (in bits) and data for how many bytes are stored per line. With this information, they can carefully stop the overflow after writing exactly 3 more pointers after the end. This overwrites the virtual table pointer and the first 4 fields of **JBIG2Bitmap**, as shown in the following image.

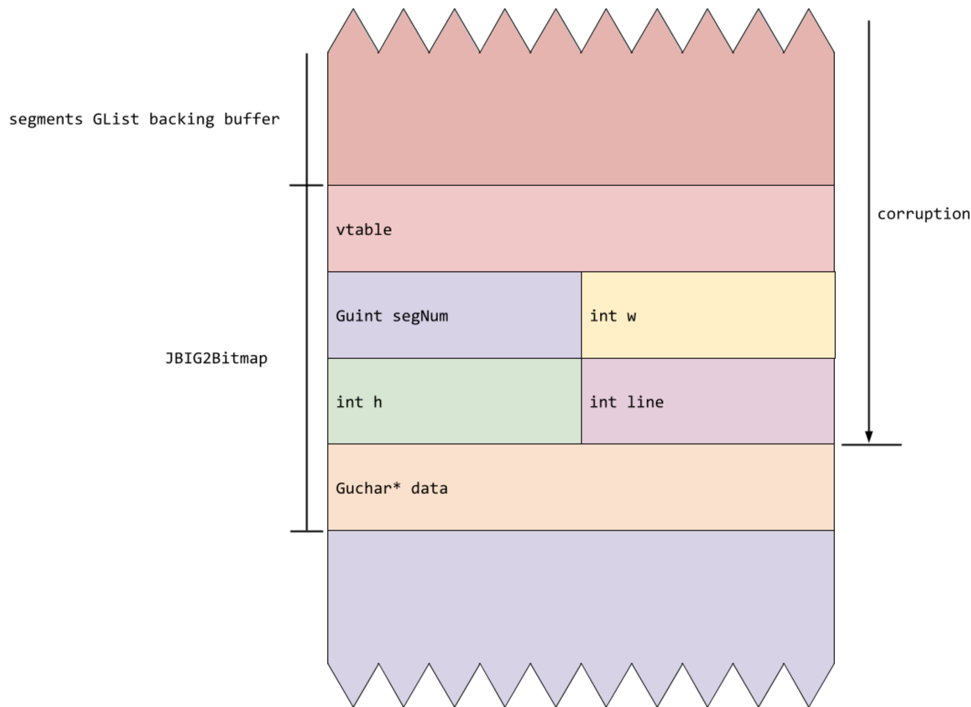

Figure 5: JBIG2 BITMAP

Due to the nature of the way Apple organizes the address space, these pointers are likely to be in the second 4 gigabytes of the virtual memory. Since iOS hardware is little endian (meaning the overwriting happens with the most significant half of the **JBIG2Bitmap** pointer first), the **segNum** and **h** field are likely to be overridden with the least-significant half of such a pointer, which would be a random value between 0x100000 and 0xffffffff. This gives the current destination page **JBIG2Bitmap** a large **h** value, which is used for bounds checking. Since it is supported to reflect the actual size of the page backing buffer, this ultimately has the effect of unbounding the buffer. Then, all subsequent **JBIG2** segment commands can read and write to memory outside of the original bounds of the page backing buffer. The heap groom also puts the current page's backing buffer right below the **syms** buffer, so that when the page **JBIG2Bitmap** is unbounded, it can read and write its own fields.

By repeating this process with the correct canvas coordinate, the attacker can write to all the fields of the page, and by choosing the right values of **w**, **h**, and **line**, they can also write to

arbitrary offsets from the buffer. Given that the attacker can access memory regions on arbitrary offsets from the current page's **JBIG2Bitmap** buffer, and since it has been unbounded, the attacker can also perform logical operations on memory at any out of bounds offset. The steps to perform this JBIG2 refinement are also very flexible, and each step can either output the bitmap and any previously created segments, or render output to the current page. If the attacker crafts the context dependent part of the refinement, it's possible to have sequences of segments where only the refinement combination operators have any effect. Thus, it is possible to perform logical operators on glyphs. Since this has been unbounded as well, it's possible to perform logical operations on any memory at an arbitrary offset from the buffer. Moreover, with just the logical operators AND, OR, XOR, and XNOR, you can develop any computable function. If these glyphs were changed to bits, then you can input a sequence of **JBIG2** segment commands that implement logical bit operations to arbitrary memory.

When JBIG2's vulnerability is exploited using the knowledge of using logic gates to operate on accessible memory, the next step is having the attacker build their own computer architecture and script that. In this exploit, there are over seventy thousand segment commands defining logical bit operations, to the extent of features such as registers, a full 64-bit adder, and a comparator. These features can search memory and perform arithmetic operations. This is not as fast as Javascript, but computationally equivalent to Javascript.

Weird Machines

A weird machine is a computational artifact in which additional code can be executed outside the original specification of the program. From a theoretical perspective, a machine can move a program from state to state, following a set of valid transitions. Illegal transitions that result in illegal states are restricted in a way defined by the software's security constraints. However, in weird machines, the system can be moved into a state that is "broken" (in this case, through memory corruption), and subsequently, the software will keep transforming the broken state into new broken states, as opposed to terminating. A broken state is defined as one that technically does not follow the constraints of the security guidelines, but still manages to enter the system and continue.

In our case, the weird machine described above was responsible for loading the next stage in the infection process, the sandbox escape. As of iOS 14, however, Apple developed BlastDoor, a significant refactoring of iMessage processing that helped improve security against these iMessage attacks. BlastDoor, a tightly sandboxed service, was responsible for parsing any untrusted data in iMessages through repetitive sanitization. Moreover, it was written in Swift, a significantly safer language with less ability to be exploited via memory corruption. Two of the

important changes made were re-randomizing the shared cache region, and using exponential throttling to slow down brute force attacks.

Address Space Layout Randomization (ASLR) has one structural weakness, that the shared cache region is randomized once per boot. Therefore, all the system libraries existed in a single pre-linked blob, and it is at the same address across all processes. This allows an attacker to remotely infer the base address of this cache by observing process crashes, and break ASLR. However, with the re-randomization of the shard cache region, the address is randomized for the target service every time it is started, which makes inferring the base address impossible, unless through brute force.

To limit the brute force technique, the interval between restarts after a crash grows exponentially each time. Thus, when an exploit originally took a few minutes, it can now take many hours.

<u>Apple's Fix</u>

In iOS 14.8, Apple patched the function by adding some bounds to avoid overflowing the **syms** buffer. This fix, although small, helped avoid the root cause of the crash and any subsequent vulnerabilities.

```
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 __ZN11JBIG2Stream17readTextRegionSegEjiijPjj:158 (181D710AC)
```

Figure 6: iOS PATCH

Chrysaor

Deriving from Pegasus, Chrysaor (Pegasus's brother in Greek Mythology) was also believed to be created by NSO Group, mainly to target Android devices. As opposed to zero-click attacks, Chrysaor was believed to be downloaded by the target, which means the attacker would have coaxed the target into perhaps clicking on a link, visiting a webpage, etc. Upon installation, the software uses framaroot exploits to escalate privileges and break into the application sandbox. Immediately after, in an effort to protect itself, the software does the following: It installs itself on **/system** to persist after any factory resets. It also disables auto-updates to maintain persistence. To collect data, it starts content observers to exfiltrate data. It has similar capabilities to Pegasus, including but not limited to: audio/video recording, recording data from SMS/Email applications, keystroke logging, etc. There exists an antidote file /sdcard/MemosForNotes that removes Chrysaor from the device. More research is yet to be done on Chrysaor, but for now, it poses a similar threat to Android users as Pegasus does to iOS users.

Conclusion:

From the advanced level of attack, we can see the technologies associated with Pegasus are extremely threatening to targeted iOS users. Pegasus jailbreaks the device by sending a PDF file labeled as a .gif file via iMessage. Then, as this file is parsed, a buffer that contains segments of bitmaps is overflowed using integer overflow. By carefully grooming the objects around this, the buffer is unbounded which allows access into all the heap memory. Finally, a weird machine built on logical operators is run in order to escape the sandbox and install the malware onto the system. These attacks are extremely powerful due to their discrete nature and their persistence. Since these attacks are often targeted towards persons of political interest, average iOS users need not worry—but it's still a good idea for civilians to be aware of the cyberthreats to which they may be susceptible.

It is important to continuously develop a plan to defend against such threats in order to protect users. Obviously, individuals are not aware of Pegasus attacks due to their discrete nature. Best measures in order to improve security of these devices would be maintaining regular security assessments, rigorous testing, and regular updating. Nevertheless, it is important for users to be aware of these attacks, and keep an eye out for strange behavior.

Works Cited

15, B. M. J. S., Authors, Analyst, M. J. T., Jin, M., Analyst, T., Us, C., & Subscribe. (2021, September 15). *Analyzing pegasus spyware's zero-click iPhone Exploit Forcedentry*. Trend Micro. https://www.trendmicro.com/en_vn/research/21/i/analyzing-pegasus-spywares-zero-click-iphone-exploit-forcedentry.html

Christian J. D'Orazio, Martini, B., Quick, D., Barmpatsalou, K., D'Orazio, C., Fan, Y., Daryabar, F., Shariati, M., Rahman, N. H. A., & Azfar, A. (2016, November 17). *Circumventing IOS security mechanisms for APT forensic investigations: A security taxonomy for cloud apps*. Future Generation Computer Systems. https://www.sciencedirect.com/science/article/pii/S0167739X16305647

A deep dive into an NSO zero-click iMessage Exploit: Remote Code Execution. A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution. (n.d.). https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

Espósito, F. (2021, January 29). *Apple adopts new "Blastdoor" security system on IOS 14 to reinforce Imessage Integrity*. 9to5Mac. https://9to5mac.com/2021/01/28/apple-adopts-new-blastdoor-security-system-on-ios-14-to-reinforce-imessage-integrity/

An investigation of Chrysaor malware on Android. Android Developers Blog. (n.d.). https://android-developers.googleblog.com/2017/04/an-investigation-of-chrysaor-malware-on.html

NSO Group - Cyber Intelligence for Global Security and stability. (n.d.-a). https://www.nsogroup.com/

Pegasus spyware: A vulnerable behaviour-based attack system | IEEE ... (n.d.-b). https://ieeexplore.ieee.org/document/10212163

Privatized espionage: NSO Group Technologies and ... - Wiley Online Library. (n.d.-c). https://onlinelibrary.wiley.com/doi/full/10.1002/tie.22321

The rise and fall of NSO Group. Forbidden Stories. (n.d.). https://forbiddenstories.org/the-rise-and-fall-of-nso-group/

Ryan. (n.d.). *A deep dive into an NSO zero-click iMessage Exploit: Remote Code Execution*. A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html

Shivi Garg, SalernoS., ZdziarskiJ., KumarR., QamarA., GargS., O'DeaS., TungLiam, ClementJ., HidhayaS.F., ZhangL., AhvanooeyM.T., TalalM., ShresthaB., LeeS., RashidiB., … WangJ. (2021, February 13). *Comparative analysis of Android and IOS from Security Viewpoint*. Computer Science Review. https://www.sciencedirect.com/science/article/abs/pii/S1574013721000125

Technical analysis of pegasus spyware - lookout. (n.d.-d). https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf

View of cyber security regimes and the violation of international law in the context of pegasus controversy: Pakistan Journal of International Affairs. View of CYBER SECURITY REGIMES AND THE VIOLATION OF INTERNATIONAL LAW IN THE CONTEXT OF PEGASUS CONTROVERSY | Pakistan Journal of International Affairs. (n.d.). https://www.pjia.com.pk/index.php/pjia/article/view/422/300