Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets

Jan Ruge and Jiska Classen, Secure Mobile Networking Lab, TU Darmstadt; Francesco Gringoli, Dept. of Information Engineering, University of Brescia; Matthias Hollick, Secure Mobile Networking Lab, TU Darmstadt

Writer: Akib Jawad Nafis

1 Introduction

Radio frequency protocols or implementations have always been a huge target for attackers. The reason behind this would be a huge attack surface. An attacker can attack wireless protocols even before it is connected to the network. Most of the wireless protocol implementation is closed source. As a result fuzzing became the best way to find bugs in these implementations. Protocol fuzzing over the air is pretty slow. We have to depend on the wireless transmission time. At the same time there can be interference while fuzzing over the air. Dependency on physical devices, limitation while repeating an experiment, complexity in debugging also contributes to the issues of over the air fuzzing. Making the wireless fuzzing faster at the same time less clunky is a great research problem.

1.1 Wireless Fuzzing of Bluetooth

One of the most widely used RF protocol implementation would be Bluetooth. Almost all of the smartphones and portable computers today has a Bluetooth chip in it. At the same time security of Bluetooth has always been kind of questionable. Bluetooth stack is divided on two parts. Host (Operating System of the device holding that Bluetooth chip) and Bluetooth Controller chip. These two in connected with a layer named HCI (Host Controller Interface). Software in the Bluetooth controller chip is called firmware. Firmware of a Bluetooth chip or any wireless chip is closed source. Hence it is hard to debug but vulnerabilities residing in the firmware can be catastrophic. At the same time fixing those vulnerabilities after deployment is another huge problem. Because firmware resides in ROM chip of the hardware, not easy to update. Fix have to come the chip vendor itself. Some vulnerability in the firmware can be remain hidden even from the operating system. As some portion of the Bluetooth hardware doesn't require any kind of interaction with the host stack. So fuzzing the Bluetooth firmware to find out bugs before an attacker choose to exploit them is a great idea. Combining these with the idea to improve over the air fuzzing is our goal in this project. We choose broadcom Bluetooth firmware to fuzz as it is widely used.

1.2 Prior Work

Fuzzing Bluetooth protocol has been mostly limited to fuzzing the host stack. Bluetooth Firmware has not been fuzzed public prior to this work. Prior to this Bluetooth Firmware research was mostly about extending the capability of the chip. There were research about security of the Bluetooth Firmware but it was manual analysis. btlejack [**btlejack**] extends capability of the BLE at the same describing man in the middle attack. InternalBlue [**DBLP:journals/corr/abs-1905-00631**] used reverse engineering of Bluetooth Firmware to read/manipulate low layer frames. It also discovers a bug in the Broadcom chip. Over the air fuzzing has been done in deepsec [**deepsec**] but it focuses on host part of the Bluetooth stack. Other fuzzing efforts was based on drivers

and operating systems. Syzkaller[**syzkaller**] supports fuzzing HCI in linux. Apart from broadcom chip, Marvel Avastar[**marvel-avastar**] WiFi chip was fuzzed using afl-unicorn[**afl-unicorn**]. TriforceAFL[**TriforceAFL**] uses similar QEMU based fuzzing but it modify the QEMU itself for other supports. On the other hand Frankenstein here uses QEMU as a userspace program for support it uses hooking the firmware. LTEFuzz [**8835363**] fuzzes LTE network over the air and found vulnerabilities in core network components.

2 Approaches Taken

To solve problems of the over the air fuzzing authors took a new approach. It is emulating the firmware to create a virtual chip. By emulation authors reduce latency from host to chip communication. They didn't have to depend on another chip to send packets over the air. They were able to cut down the transmission time by generating packets in the device itself. Emulated virtual chip can be connected to the Bluetooth host side of linux system. Thus creating a full system emulation environment.

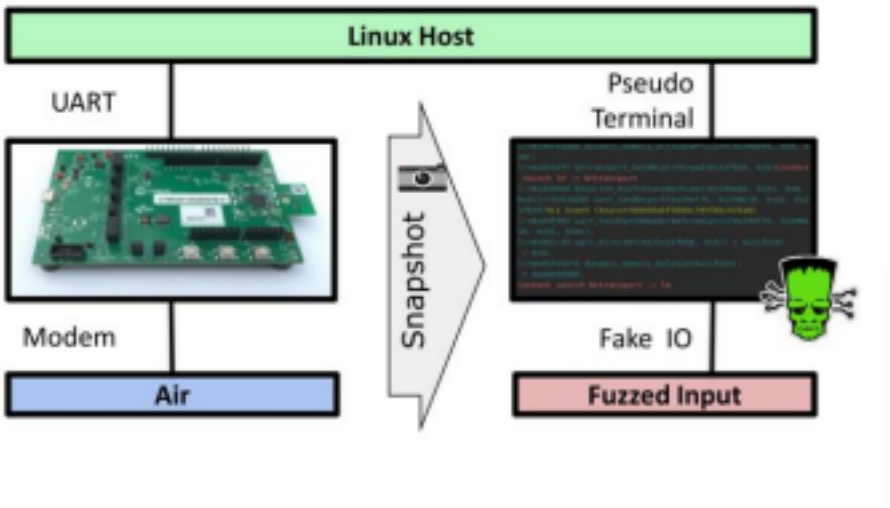

Figure 1: Emulation Based Fuzzing

2.1 Emulating the firmware

Authors pulled snapshot from the physical device and then used it as a firmware to emulate in QEMU. But a snapshot can't be directly emulated. It is just a binary. A elf executable file must be generated to execute it in the QEMU environment. That elf executable need some more support.

This frankenstein will run as a user-space program it doesn't support any interrupt or timer itself. But Bluetooth firmware has it's own interrupt and timer handlers. These interrupt and timer handlers need to be implemented manually for the virtual chip. It could have been done in the QEMU itself, but then that QEMU emulation will not be user-mode emulation. Along with this support debug symbols and coverage hooks must be included in the firmware for fuzzing. Including all of those a virtual chip is generated.

2.2 Connecting Virtual Chip to the Host

To connect this virtual chip to the linux host Pseudo Terminal is used. Pseudo terminal is basically a pipe to connect two programs. Master end of the pipe belongs to the virtual chip and slave end of the pipe belongs to the host side of Bluetooth stack.

2.3 Random packet generation

To generate inputs packets for the firmware a virtual modem is created. Virtual modem creates various types of packet to trigger various portion of the firmware codebase.

2.4 Mutating inputs while generating inputs

Frankenstein uses a different type of mutation techniques. Typical fuzzers uses input as a large binary object and modify it. Frankenstein modified this process slightly. Instead of considering whole packet as a binary object, it took two parts of the packet (Sequence No and Data) and modified them separately. This method results in better code coverage[**??**]. Classic blob (grey line)

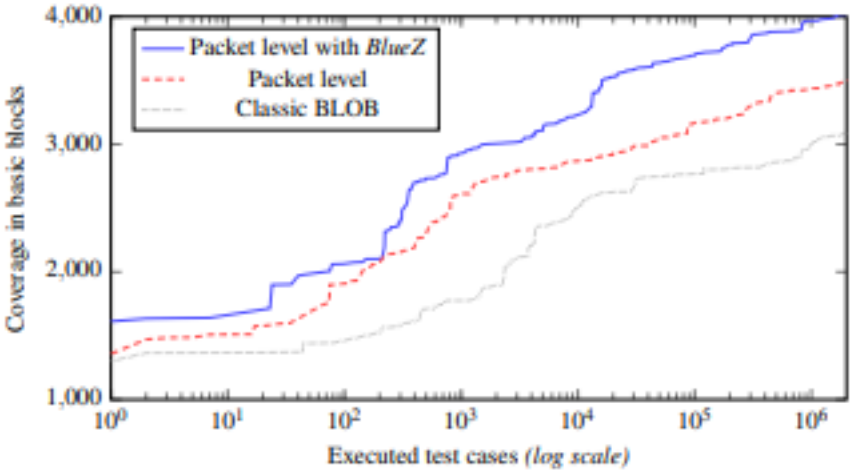

Figure 5: LMP fuzzing strategy comparison.

Figure 2: Code coverage

represents mutation of the packet as a binary blob. Packet Level (red line) is modification to packet data only. Packet Level with BlueZ includes mutation of both sequence no and packet data.

2.5 Evaluation based of bugs found

Frankenstein discovered two types of bug in the Bluetooth firmware. One is Remote Code Execution based and another is heap corruption based. Types of Bugs and their CVEs are included in the list below.

2.6 Evaluation based on novelty and speed

Since it is the first systemic approach to find bugs in the bluetooth firmware via emulation based fuzzing, it has novelty in it. Also it increases efficiency than over the air fuzzing.

RCE bugs	Heap Corruption Bugs
Link Key Extraction.	Device Scanning EIR (CVE-2019-11516).
Disabling Wifi by writing a specified value while testing in a wifi/bluetooth combo chip. (CVE-2019-15063).	Any BLE Packet (CVE-2019-13916).
	Any ACL Packet (CVE-2019-18614).
	BlueFrag (CVE-2020-0022).

Table 1: Vulnerabilities and CVE's found by frankenstein

3 Deliverable

Two types deliverable while reproducing this work.

- Reproducing their emulations.
- Reproducing CVE using the emulator.

3.1 Reproducing their emulations

Systemwise build will generate 8 emulations. Emulating all modules will be time consuming. I'll

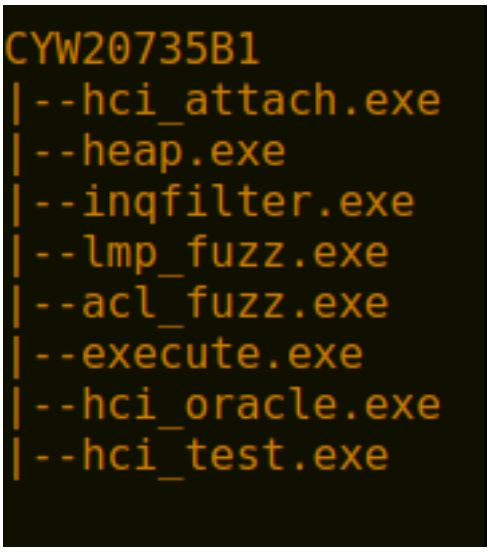

Figure 3: emulations

try to emulate and report my finding form the execute.exe, heap.exe, hci_attach.exe, inqfilter.exe, lmp_fuzz.exe, acl_fuzz.exe. [It might change since I don't know which emulation takes how much time]

3.2 Reproducting CVEs

I will reproduce two CVEs.

- Device Scanning EIR (CVE-2019-11516)

- Any BLE Packet (CVE-2019-13916)

I won't be able to to reproduce some of the CVE's due to device constraint.

- Any ACL Packet (CVE-2019-18614) [Reproducing this experiement bricked a module of authors. Also lack of time :( ]
- BlueFrag (CVE-2020-0022). [Need an Android 8.0-9.0 device wth this specific bluetooth chip to follow the steps.]
- Disabling Wifi by writing a specified value while testing in a wifi/bluetooth combo chip. (CVE-2019-15063) [It's done on wifi/bluetooth combo chip. I don't have any neither physical nor virtual.]

4 Experiments

As per proposal I emulated 5 of the 8 emulations mentioned in the project. Also reproduced two CVEs from this project. Initially I'll describe experimental setup. Then I'll go through the emulations. Later I'll describe how CVE's are reproduced and what triggered those bugs.

4.1 Experimental Setup

Authors goal was to execute Bluetooth firmware in a virtual environment so that they can fuzz the firmware easily. To do this, authors copied all the memory segments of a running firmware. Authors copied the memory segment with a script. Authors provided memory segments of the firmware CYW20735 Bluetooth evaluation board. Patched the firmware to execute it without the actual hardware. A hooking mechanism which mainly use c-construct is used for patching process. Then run the virtual firmware to test various portion of the firmware. To check various portion of the firmware authors created multiple emulations which will execute in the virtual environment.

4.2 Emulation: execute.exe

In this emulation authors simply confirmed they can interact with virtual firmware executing in QEMU. They also confirmed firmware is executing it's three threads(bttransport, lm and idle) correctly. When both bttransport and lm thread is waiting for events firmware will moe to idle thread. When firmware reaches that idle thread authors hooked their external code to check that they can interact with the firmware. From the emulation **??** we can see firmwre thread switching and once it reaches idle state as per external code it exits execution.

4.3 Emulation: hci_attach.exe

In this emulation author confirmed that the firmware can be executed with the Bluetooth stack of operating system. They used linux bluetooth stack Bluez to connect the firmware. After successful connection it will act as a complete virtual Bluetooth device. To check functionality of the virtual bluetooth device, I tried scanning for bluetooth devices**??**. Random packets(containing bluetooth device addresses) are feed to the virtual device via terminal.

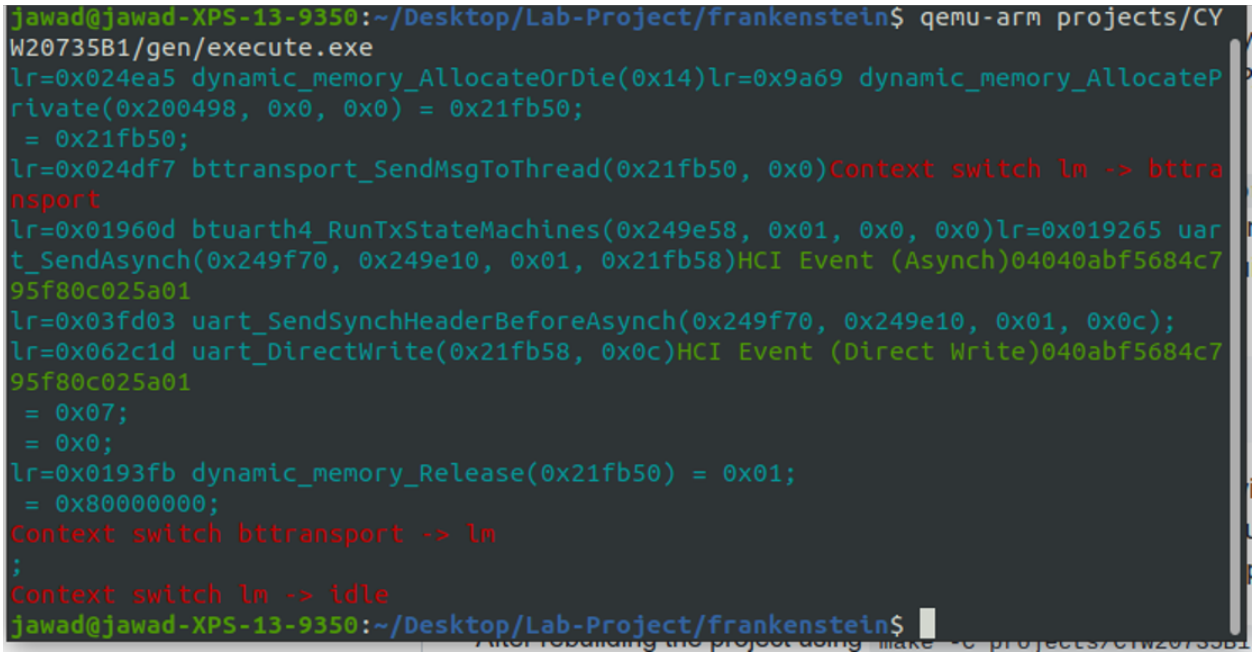

Figure 4: Emulating execute.exe

4.4 Emulation: lmp_fuzz.exe

In this emulation, authors executed link management protocol implementation of the firmware. Random packets are feed to the firmware by hooking a function of the firmware `lm_LmpReceived()`. In a real device this function is called when a new link management packet has been received. Fuzzing this experiment didn't trigger any bug. Figure **??** shows the relevant function calls while processing random link management packets. This experiment also emulated the portion when a lmp packet creates a hci event at host end. Generated hci command is sent to the host by using `lm_sendCmd()` function.

4.5 Emulation: acl_fuzz.exe

Similar to link management protocol fuzzing, authors fuzzed asynchronous connection-less packet transfer process. Authors feed the system random payload packet to observe the packet sending and receiving process. Fuzzing this experiment didn't trigger any bug neither in authors experiment nor in my experiment. Goal was to check firmware is executing correctly in the virtual environment. Figure **??** and **??** shows the relevant function calls while processing random acl packets.

4.6 Reproducing CVE-2019-11516

It is a bug which causes the firmware to crash while searching for other bluetooth devices nearby. When a bluetooth device is scanning for other devices, it sends a packet. This packet is called inquiry packet. When nearby bluetooth devices wants to respond to this inquiry they can send a response packet. In some cases, these nearby bluetooth devices sends an extended inquiry response. Problem is a malformed extended inquiry response will crash the inquiring firmware. Here malformed response means RFU bits of the inquiry response are anything but 0.

In my experiment, I tried to scan for devices with the virtual bluetooth device. While fuzzer is feeding the virtual bluetooth device with random inquiry responses. After sometimes firmware

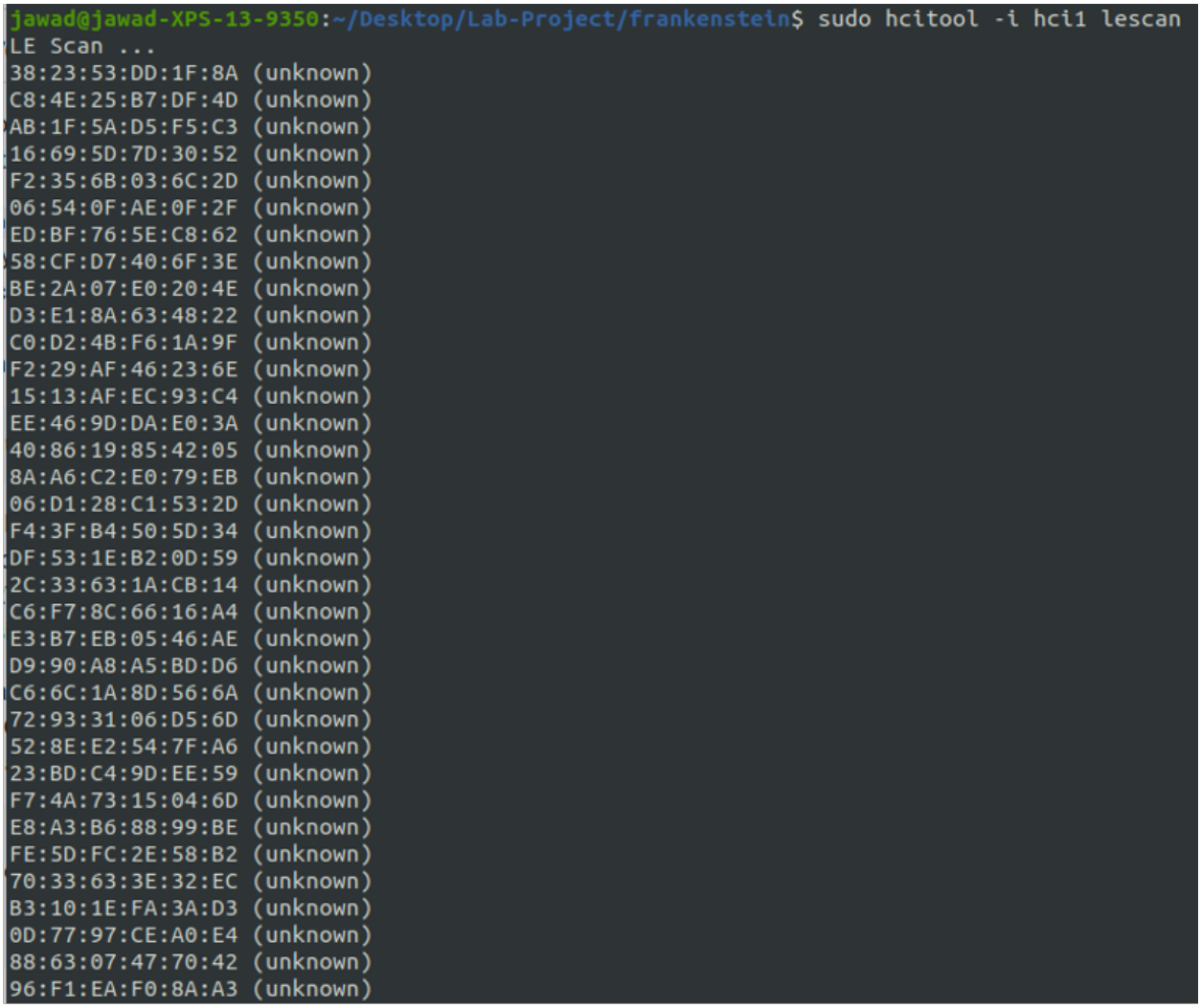

Figure 5: Emulating hci_attach.exe

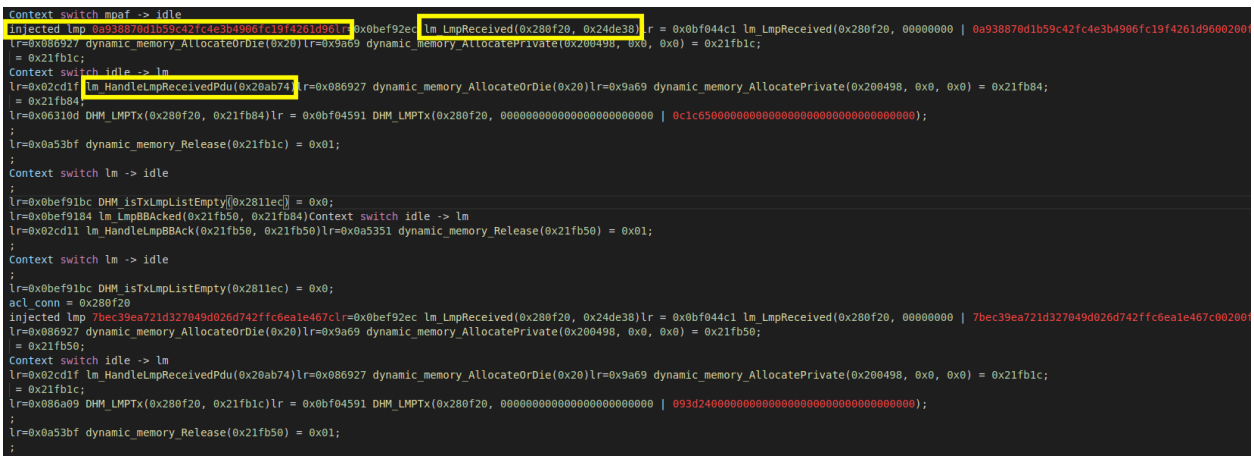

Figure 6: Emulating lmp_fuzz.exe

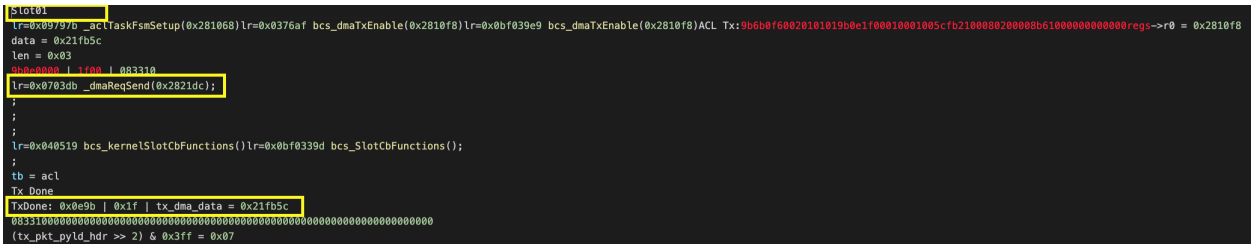

Figure 7: ACL packet sending

Figure 8: ACL packet receiving

of the device which was scanning crashed. Because a inquiry response had the RFU bits set. And while processing the response in the function `inqfilter_registerBdAddr()` firmware crashed. Figure **??** shows the experiment where in the right terminal virtual device is scanning and the left terminal shows what is happening inside the firmware.

4.7 Emulation: inqfilter.exe

This emulation is created to show the bug mentioned in the previous section. In this emulation authors simply calls the function `inqfilter\_registerBdAddr()` with a crafted inquiry response. Results are shown in the figure **??**

4.8 Reproducing CVE-2019-11916

When a bluetooth device is trying to connect to a Bluetooth Low Energey(BLE) device or exchanging packets with a BLE device, This bug might trigger. So the problem is when the other BLE device sends a packet which has a PDU more than 252 bytes. In the receiving firmware side, it causes a heap corruption error.

In my experiment, I tried to connect to a BLE device with the virtual bluetooth device. While fuzzer is feeding the virtual bluetooth device with random BLE packets. At a point a packet with packet length `oxff` caused the crash. Figure **??** shows the experiemnt. In the right side our virtual bluetooth device is trying to connect a ble device with the command `lecc`. Left side shows what is happening inside it's firmware.

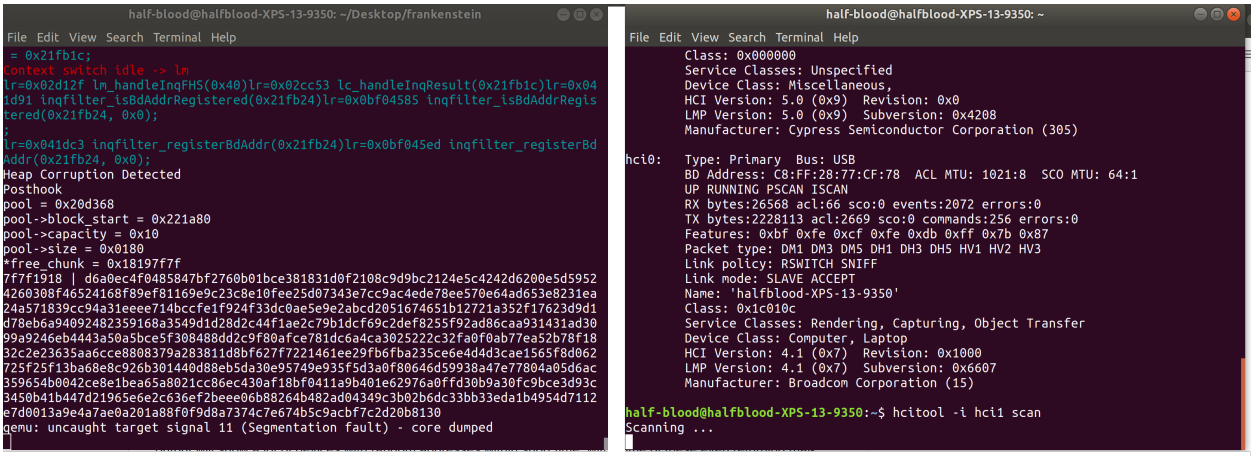

Figure 9: Reproducing CVE-2019-11516

```
erBdAddr(0xfffef090, 0x0);
;
i = 0x2e
lr=0x0bef91a4 inqfilter_registerBdAddr(0xfffef090)lr=0x0bf046e1 inqfilter_regist
erBdAddr(0xfffef090, 0x0);
Heap Corruption Detected
Posthook
pool = 0x20d368
pool->block_start = 0x221a80
pool->capacity = 0x10
pool->size = 0x0180
*free_chunk = 0x412f7f7f
7f7f2f41 | 414141410485847bf2760b01bce381831d0f2108c9d9bc2124e5c4242d6200e5d5952
4260308f46524168f89ef81169e9c23c8e10fee25d07343e7cc9ac4ede78ee570e64ad653e8231ea
24a571839cc94a31eeee714bccfe1f924f33dc0ae5e9e2abcd2051674651b12721a352f17623d9d1
d78eb6a94092482359168a3549d1d28d2c44f1ae2c79b1dcf69c2def8255f92ad86caa931431ad30
99a9246eb4443a50a5bce5f308488dd2c9f80afce781dc6a4ca3025222c32fa0f0ab77ea52b78f18
32c2e23635aa6cce8808379a283811d8bf627f7221461ee29fb6fba235ce6e4d4d3cae1565f8d062
725f25f13ba68e8c926b301440d88eb5da30e95749e935f5d3a0f80646d59938a47e77804a05d6ac
359654b0042ce8e1bea65a8021cc86ec430af18bf0411a9b401e62976a0ffd30b9a30fc9bce3d93c
3450b41b447d21965e6e2c636ef2beee06b88264b482ad04349c3b02b6dc33bb33eda1b4954d7112
e7d0013a9e4a7ae0a201a88f0f9d8a7374c7e674b5c9acbf7c2d20b8130
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
```

Figure 10: Emulating inqfilter.exe

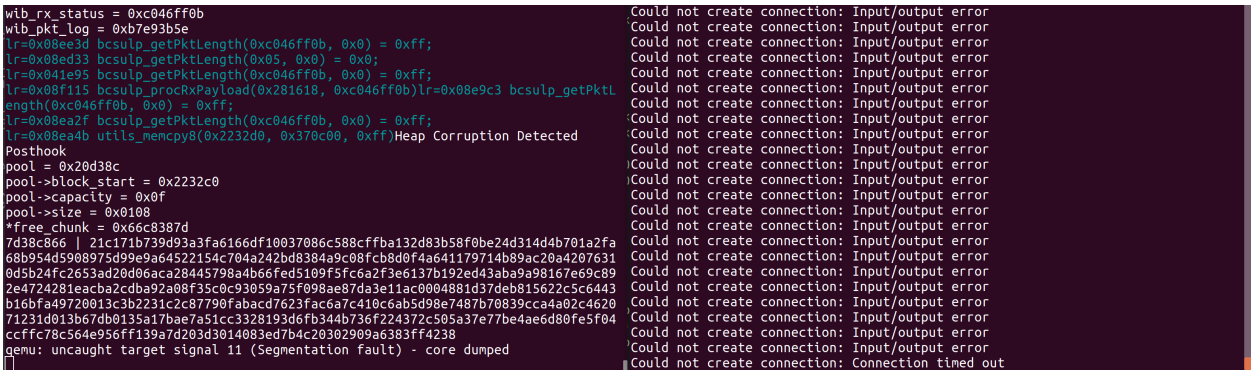

Figure 11: Reproducing CVE-2019-11916

5 Discussion

As per deliverable mentioned above **??**, I completed all the experiment I mentioned in the proposal. I also checked one more emulation `hci_oracle.exe` which I didn't mention in the proposal. Due to time limit and device limitation I couldn't complete all the experiment mentioned in the article. But I would say I completed most part (6 emulations out of 8 and reproducing 2 CVEs) from the article. Focus of this article was about emulating a firmware to complete wireless fuzzing. I would focus was achieved completely.