RESEARCHING MARVELL AVASTAR WI-FI: FROM ZERO KNOWLEDGE TO OVER-THE-AIR ZERO-TOUCH RCE

Denis Selianin
Agenda

Broadly:
It is all about how device security can be completely compromised using component vulnerabilities.

Specifically:
• How Wi-Fi devices works/Attack surface of Wi-Fi devices
• RE RTOS ThreadX
• Instrumentation and fuzzing of Wi-Fi firmware
• Exploitation of vulnerabilities on Wi-Fi SoC
• Escalation to the Application Processor (AP)
Previous research

• Series of blog posts Google Project Zero by Gal Beniamini (starting from April 2017)
• Black Hat USA 2017 - Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets
• SEEMOO lab projects (not actual vulnerability research)
• Some mobile pwn2own baseband exploits and write-ups (focused on baseband)
  • https://github.com/comsecuris/shannonRE
Where Marvell Avastar Wi-Fi can be found

- Sony PlayStation 4, PlayStation 4 Pro
- Microsoft Surface, Surface Pro, Surface laptop, Xbox One
- Samsung Chromebook, some smartphones like Galaxy J1
- Valve SteamLink, and other devices...
How it works

- Difference between FullMAC and SoftMAC
How it starts up

Operating System driver startup

File from filesystem

Uninitialized

Initialized
Researched device

- Marvell Avastar 88W8897
  - Steamlink Wi-Fi
    - GNU/Linux
    - mlan + mlinux kernel modules
  - Wi-Fi core - ARM946 core
  - Wi-Fi + Bluetooth + NFC COMBO
Firmware internals of Marvell FullMAC Wi-Fi

- Linux-firmware package or repo is a source of this blobs
- RAM image files – structure from driver
- IDA loader
- Contains several memory regions configured by MPU.
- Missing ROM?

```c
struct mwifiex_fw_header {
    __le32 dnld_cmd;
    __le32 base_addr;
    __le32 data_length;
    __le32 crc;
} __packed;

struct mwifiex_fw_data {
    struct mwifiex_fw_header header;
    __le32 seq_num;
    __le8 data[1];
} __packed;
```

```
<table>
<thead>
<tr>
<th>Offset (h)</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000001</td>
<td>1C 30 FF E0 1C F0 EF FF E0 1C F0 EF FF E0</td>
</tr>
<tr>
<td>00000002</td>
<td>00 20 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000003</td>
<td>00 20 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000004</td>
<td>14 22 00 00 00 22 00 00 00 34 22 00 00 00 44 22</td>
</tr>
<tr>
<td>00000005</td>
<td>50 22 00 00 00 00 00 00 00 00 4F 0F 07 00 00</td>
</tr>
<tr>
<td>00000006</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000007</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000008</td>
<td>0B 0A 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00000009</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0000000A</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0000000B</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0000000C</td>
<td>00 0C 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
```

- `dnld_cmd`
- `base_addr`
- `data_length`
- `crc`
Marvell Wi-Fi device interaction with AP

- Linux driver
  - GNU licensed mwifiex
  - Marvell proprietary mlan+mlinux

- API and events – command packets
  - Serialization/Deserialization to internal format

- Versions of firmware and driver depends on a chip and interconnection bus (sd8897.bin vs pci8897.bin)

- Higher layer packets encapsulated in a lower layer
  - For example in SDIO RW or PCI bus TLP
Marvell Wi-Fi device interaction with AP. cont.
Firmware API implemented in driver

- READ/WRITE functions of SoC memory
- Extended version info from firmware (like “w8897o-B0, RF8XXX, FP68, 15.68.7.p206” for SteamLink)
- Wi-Fi related stuff (authentication, association, scanning...)

- Some of them can be accessed from the usermode
- It is much easier to RE firmware dump
Post-mortem analysis of firmware crash

Operating System driver

Request
Timeout

Wi-Fi chip

Dumping memory
Firmware debug crash - differences

• mwifiex
  • PCI DUMP – to devcoredump a linux device
  • Contains FULL Wi-Fi SoC memory dump
  • Format similar to a firmware image in the filesystem
  • Additional driver info and statistics

• mlan + mlinux
  • SDIO DUMP – directly to host OS filesystem
  • Contains SEVERAL memory regions (ITCM, DTCM, SQRAM, ...)
  • RAW binary format – separate files
  • Additional driver info and statistics
Starting RE of firmware dump

- No symbols (approx. < 10 strings)
- Approx. 5K functions. Some of them exceeds limits of IDA ( > 1000 BB)
- No information about RTOS
- ARM code. Most is thumb code
- Only interrupt vectors
- We can find MPU initialization
  - Identify boundaries of memory regions
  - Memory regions are RWX
### Firmware memory layout for 88W8887

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>ITCM</td>
</tr>
<tr>
<td>0x00000000</td>
<td>DTCM</td>
</tr>
<tr>
<td>0x04000000</td>
<td>UNKNOWN</td>
</tr>
<tr>
<td>0x04000000</td>
<td>HEAP + THREAD</td>
</tr>
<tr>
<td>0x0C000000</td>
<td>STACK</td>
</tr>
<tr>
<td>0x0C000000</td>
<td>MAIN CODE + ROM</td>
</tr>
<tr>
<td>0x80000000</td>
<td></td>
</tr>
<tr>
<td>0x90000000</td>
<td></td>
</tr>
</tbody>
</table>
RE of Firmware

- Use full memory dumps instead of loaded image FW
  - You can get runtime structures
- Appears to be a ThreadX – based bare-metal firmware
- Recover ThreadX runtime structure from live memory dump
- Recover RTOS tasks + stacks
  - You can get entry points !!! (with names in case steamlink firmware)
- Recover block and byte pool memory layout
  - Essential for hunting bugs
ThreadX RTOS

• One of the most popular RTOSes
  • Over several billions deployments

• Closed sourced, however leaked sources for earlier versions can be found

• Provides basic API and services
  • Thread scheduling
  • Counting semaphores
  • Mutexes
  • Block and byte pool memory management
  • Timers
  • …
ThreadX runtime structures

- Contain signature fields, by which they can be identified in memory dump
- Also helps to identify RTOS functions (because of ARM constant handling)
# ThreadX runtime objects in Steamlink Wi-Fi firmware

<table>
<thead>
<tr>
<th>Object</th>
<th>Name</th>
<th>Entry point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Idle</td>
<td>0xFFD06479</td>
</tr>
<tr>
<td>Thread</td>
<td>MAC Tx</td>
<td>0xFFD50C39</td>
</tr>
<tr>
<td>Thread</td>
<td>MAC Tx Notify</td>
<td>0xFFD55B2F</td>
</tr>
<tr>
<td>Thread</td>
<td>MAC Mgmt</td>
<td>0xFFD13E55</td>
</tr>
<tr>
<td>Thread</td>
<td>CB Proc</td>
<td>0xFFD24859</td>
</tr>
<tr>
<td>Thread</td>
<td>IccTask</td>
<td>0xFFD066D5</td>
</tr>
<tr>
<td>Timer</td>
<td>SleepConfirmTmr</td>
<td>0xFFD1E055</td>
</tr>
<tr>
<td>Timer</td>
<td>AP_NullPktDoneTmr</td>
<td>0xFFD1DC55</td>
</tr>
<tr>
<td>Timer</td>
<td>NullPktDoneTmr</td>
<td>0xFFD1DC55</td>
</tr>
<tr>
<td>Queue</td>
<td>TxMgmt80211MsgQ</td>
<td>-</td>
</tr>
<tr>
<td>Queue</td>
<td>MacMgmtSMEMsgQ</td>
<td>-</td>
</tr>
<tr>
<td>Queue</td>
<td>TimerCbMsgQ</td>
<td>-</td>
</tr>
</tbody>
</table>
RE of firmware memory dump

- Still large and opaque binary
- Need to recover data flows inside firmware
  - Identify frame parsing routines
- Need basic firmware instrumentation to do so
Firmware instrumentation

- Extremely limited resources on Wi-Fi SoC
  - Only several Kbytes of free memory available
- However, we can hook a single function (splicing)
- We can replace pointers for some debug-or-log-like routines
- Can trace block pool allocation/deallocation
- We can even instrument entire code regions (not so big) with thumb function calls (like DBI with function-level granularity)
- All of this can be accomplished using READ/WRITE firmware API functions and extended version info API
Instrumenting firmware using debug callbacks

• Though ThreadX block pool management routines are located in ROM, firmware uses wrappers, which contain debug callback routine
Detour all calls in memory region to the instrumentation tool.
Instrumentation stub

1. Call custom instrumentation routine
2. Lookup and call original procedure by saved LR
3. Return to saved LR location
1. SAVE ORIGINAL LR
2. PREPARE ARGUMENTS FOR CUSTOM TOOL
3. CALL TOOL
6. LOOKUP FUNCTION ADDRESS IN TABLE
7. PREPARE ARGUMENTS FOR FUNCTION
8. CALL FUNCTION
11. RESTORE ORIGINAL LR
12. BX LR

4. ANALYSIS PAYLOAD
5. BX LR

9. ORIGINAL Firmware FUNCTIONALITY
10. BX LR

LOOKUP TABLE
INSTRUMENTATION STUB
DBI TOOL
ORIGINAL FUNCTION
How to achieve this?

Instrumentation workflow

Preparatory stages on AP

Patching code on the SoC
Firmware instrumentation. Code that runs on AP

- Read memory block from Wi-Fi SoC
- Disassemble it with capstone engine
- For each BL instruction
  - Get BL instruction location and target address (4 bytes)
  - Encode new BL to INSTRUMENTATION stub location on SoC (4 bytes)
  - Add entry to LOOKUP table and PATCH table
- Write PATCH table, LOOKUP table PATCHER code, INSTRUMENTATION stub and user tool to Wi-Fi SoC
- Hook extended version info function so PATCHER code will be executed, when firmware calls this function
Firmware instrumentation. Code that runs on AP. cont

Disassemble and process with capstone.

Write tables to SoC.

ADDRESS_1
BL TARGET_1

ADDRESS_2
BL TARGET_2

... MEMORY REGION FROM SOC

LR1 → TARGET_1
LR2 → TARGET_2
......

LOOKUP TABLE

ADDRESS_1 → PATCH_1
ADDRESS_2 → PATCH_2
......

PATCH TABLE

ADDRESS_1
BL TARGET_1

ADDRESS_2
BL TARGET_2

...
Firmware instrumentation. Code that runs on SoC

- Disable interrupts
- Apply patches from PATCH table to code
  - This is just replacing one BL instruction to another
- Enable interrupts
Firmware instrumentation. Code that runs on SoC cont

HOOKED EXTENDED VERSION INFO
FUNCTION (NEVER CALLED BY DRIVER)

DISABLE IRQ

APPLY PATCHES
FROM TABLE

ENABLE IRQ

CONTINUE
FUNCTION EXEC

PATCHER CODE
Firmware instrumentation/Useful tools

• Searching frame signatures in function parameters (e.g. MAC, BSSID...)
  • Identifying parsing routines
  • Also we can identify useful routines for escalation to AP

• Collection of thread context and parameters before function calls
  • Can be used for fuzzing
  • Can help RE (for example call stacks)

• ThreadX block pools state monitoring
  • Can help understand how to exploit vulnerability
Exploitation mitigations on Wi-Fi SoC

- Almost nothing
  - No ASLR
    - Very limited resources of chip
  - No DEP
    - All MPU memory regions configured as RWX
  - No stack cookies, allocator/deallocator verifications
    - Possibly by RTOS design
Hunting for bugs

- Manual
  - Hard!
- Fuzzing – still feasible using afl-unicorn fuzzer
  - Mix of AFL and QEMU mode patch applied to Unicorn emulator originally created by Nathan Voss
  - Check out materials on medium how to fuzz arbitrary code or CGC binary example
Fuzzing firmware

- Identify parsing routines and their arguments using self-made DBI
- Write fuzzer using afl-unicorn which will fuzz this routines
- Looks like an easy target
Fuzzer workflow

• MAP necessary memory regions using modified version of Unicorn
  • We have already dumped them using our tool
• Setup register context
  • Capture this one using DBI tool, or function hooking
• Read mutated input file and map it into emulator memory
  • Identify parsing routines using DBI and pass mapped memory block as function parameter
• Start code execution
  • All SET !!!
Challenges of fuzzing firmware

• It is difficult to locate and remove checksum code especially in authenticated frame handing routines.

• Out fuzzer depends on global state captured at the time when we created dump of SoC memory. Memory dump can contain saved state of global vars, block pools...etc which can prevent certain execution path to be reached by fuzzer.

• No memory access sanitization (however it can be implemented).

• Communication between RTOS tasks cannot be implemented, so some paths cannot be reached.
Results of fuzzing
Disclosure timeline

• Some bugs were founded ~4
• Vendor notified – 02 May 2018
• Submitted for ZeroNights – September 2018
• Talk selected for presentation – October 2018
• Presentation slides reviewed by Marvell – 12 November 2018
• ZeroNights conference – 21 November 2018
• Still fixing
Testing on other devices

- Different memory layout on different chips
- Different dynamic memory layout on different firmware versions
- May depend on interconnection BUS type
- Bugs are still present!
  - Compared 88W8897 firmware from linux-firmware with steamlink repo firmware
  - Compared SDIO 88W8897 with PCI 88W8997 (Samsung Chromebook)
The most interesting bug to be exploited

- The most interesting bug is the one that can be triggered during network scan
- There is no authentication
- There is no need to know which network name (SSID) victim is expecting
- **Can be triggered whether a victim is connected to network or not and without ANY user interaction (every 5 minutes in case of Marvell Wi-Fi)**
- Appears to be a ThreadX block pool overflow during network scan
ThreadX block pool overview

TX_BLOCK_POOL STRUCTURE

free list

PTR Block 1

PTR Block 2

NULL Block n
ThreadX block pool overview cont.

LINKS CAN BE REARANGED

[Diagram showing block pool with links]

- BLOCK_1
  - PTR_TO_2
  - BLOCK_2
    - PTR_TO_1
    - BLOCK_3
      - PTR_TO_4
      - BLOCK_N
        - NULL

42
ThreadX block pool allocation of block

TX_BLOCK_POOL
STRUCTURE

free list

Previous free list RED
pointer returned to caller
Exploitation – basic technique

• Relocate next block to location where some function pointers or even regular code reside

• By writing to this newly allocated block attacker can overwrite code or function pointers
Returning attacker-controlled pointer to caller

Two consequent allocation calls

Previous attacker-overwritten RED pointer returned to caller

TX_BLOCK_POOL STRUCTURE

free list

 PTR Block 1

 PTR Block 2

 NULL Block n

 PTR Block 1

 PTR Block 2

 NULL Block n

TX_BLOCK_POOL STRUCTURE

free list
Exploitation – a simpler way

- Marvell implementation of block deallocator wrapper function listed below

- Allows direct code execution after freeing block if we can overwrite metadata in the beginning of the block

```c
if ( *(unsigned __int8 *)(memory_ptr + 0x10) << 31 ) {
    pre_release_callback = *(int (__fastcall **)(int)) (memory_ptr + 0x18);
    if ( pre_release_callback )
        post_release_callback = (void (*)(void)) pre_release_callback(memo
```

return 0;
```
Exploitation – a simpler way

- BLOCK_1
  - PTR_TO_STRUCT
  - BLOCK_2
  - PTR_TO_STRUCT
  - PTR_TO_LOG

Attacker-controlled

BUSY
Exploiting Valve Steamlink

- Linux kernel 3.8.13-mrvl arm7L
- Wireless – Marvell Avastar 88W8897 chipset
- Wireless firmware version – “w8897o-B0, RF8XXX, FP68, 15.68.7.p206”
- SDIO bus
- Wireless driver – mlan + mlinux proprietary kernel modules (sd8897.ko + 8897mlan.ko)
Exploit – stage1

• Exploit RCE bug in Wi-Fi firmware and gain control over Wi-Fi SoC
  • Beacon frame spraying
    • Because shellcode from just one frame is not enough
    • Beacon frames are located at predictable location (for certain fw version)
  • Egg-Hunter execution
    • this is all what we can deliver in a single frame
  • BRANCH to sprayed code
    • remember ARM address alignment requirements
Exploit – stage1 pic

Wi-Fi SoC Memory

Sprayed Beacon frame location

Egg-hunter code execution
We can exploit vulnerabilities in host driver command packet parser to gain execution on application processor.
This is our ESCALATION attack surface
Exploit – stage2

• Prepare for escalation to application processor
  • Hook function in firmware which sends “event” packets to host
  • Craft special firmware API response packet(s) or event packet(s) which triggers vulnerability in Marvell mlan+mlinux driver
How to write stage2 shellcode

• Information on structure of event packets can be obtained from driver source
• We can write a DBI tool to search for this structures in Wi-Fi SoC memory
Analysing linux driver

- Large project (somewhat ~150 KLOC)
- However driver has a good debug functionality that can be configured at runtime
  - Trace functions called in driver
  - Hex dump packets from Wi-Fi SoC and more
Using libtooling to analyze big amount of source code

• Write your own tool using AST information from libtooling to identify potential dangerous code
  • memcpy with variable length
  • memcpy to stack buffers

• Collect information from your tool and manually analyze it

• ~2 days to code, ~1 min to parse, ~20 minutes to analyze logs and search for vulnerability
Exploit – stage3

• Execution on host AP in kernel mode
  • Preparatory stages (ROP) – steamlink uses kernel without ASLR
  • **We need preparatory stages for mitigating ARM I/D-cache incoherency**
  • Actual payload execution in kernel mode of Application Processor
Exploit requirements

• HARDWARE
  • Wi-Fi dongle with monitor
    mode and frame injection
    capabilities
  • ALFA networks appears to be
    the best in injecting frames and
    doing it **FAST** (rtl8287 chip)

• SOFTWARE
  • Kali GNU/Linux
  • Scapy python framework
Conclusions

- Wireless devices expose **HUGE** attack surface
- Usually no exploitation mitigation present on wireless SoC
- Device drivers may expose **WIDE** attack surface for escalation from a device to host application processor
- Methods described in this research can be applied to similar devices like Broadcom Wi-Fi and smartphone baseband processors firmware
- Will publish full exploit write-up, exploit itself, tools and whitepaper as soon as fix will be available
THANKS FOR ATTENTION

@author