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

Denis Selianin

ZERO NIGHTS 2018

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

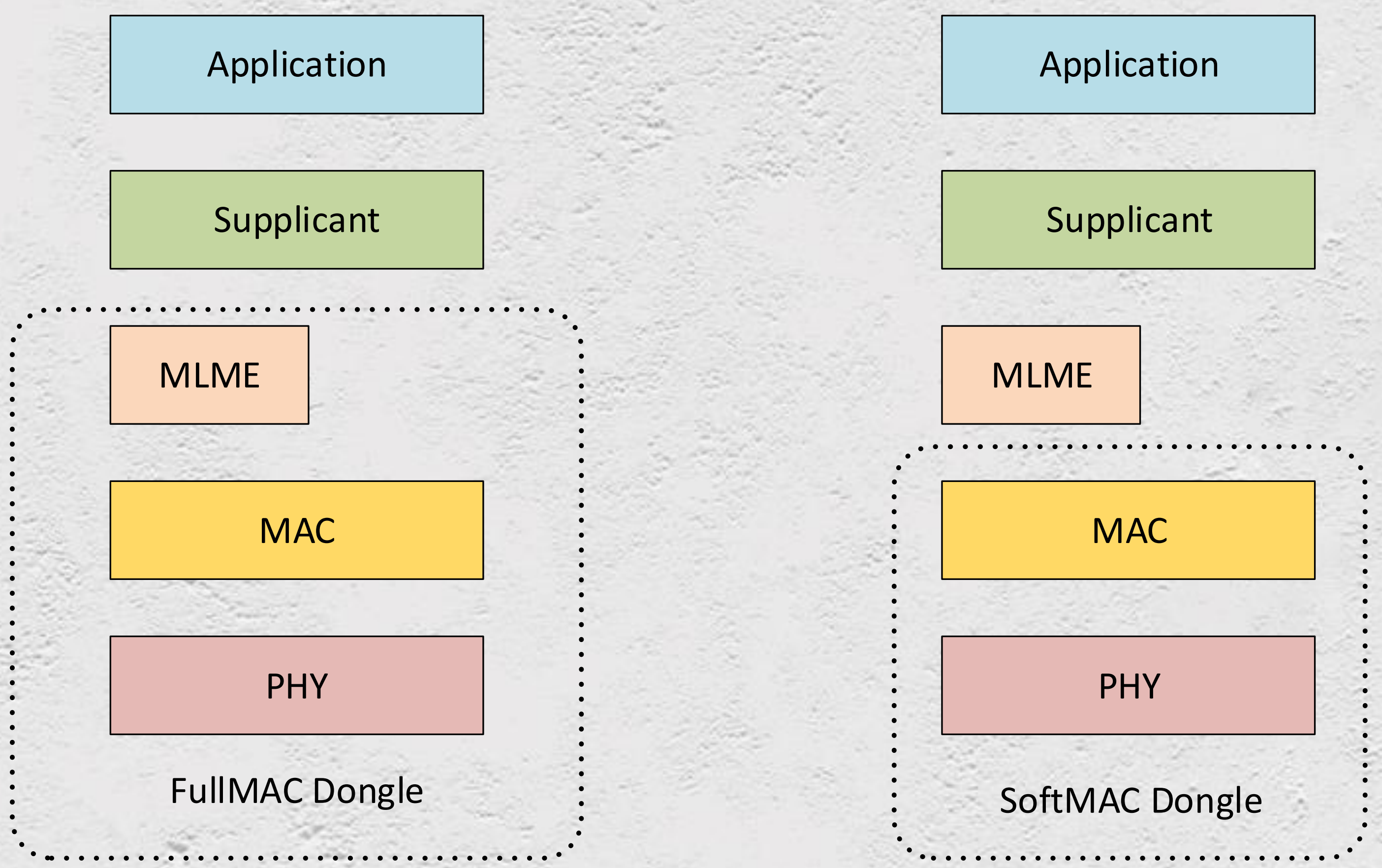
- Series of blog posts Google Project Zero by Gal Beniamini (starting from April 2017)
- Black Hat USA 2017 - Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets
- SEEMOO lab projects (not actual vulnerability research)
- Some mobile pwn2own baseband exploits and write-ups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

Where Marvell Avastar Wi-Fi can be found

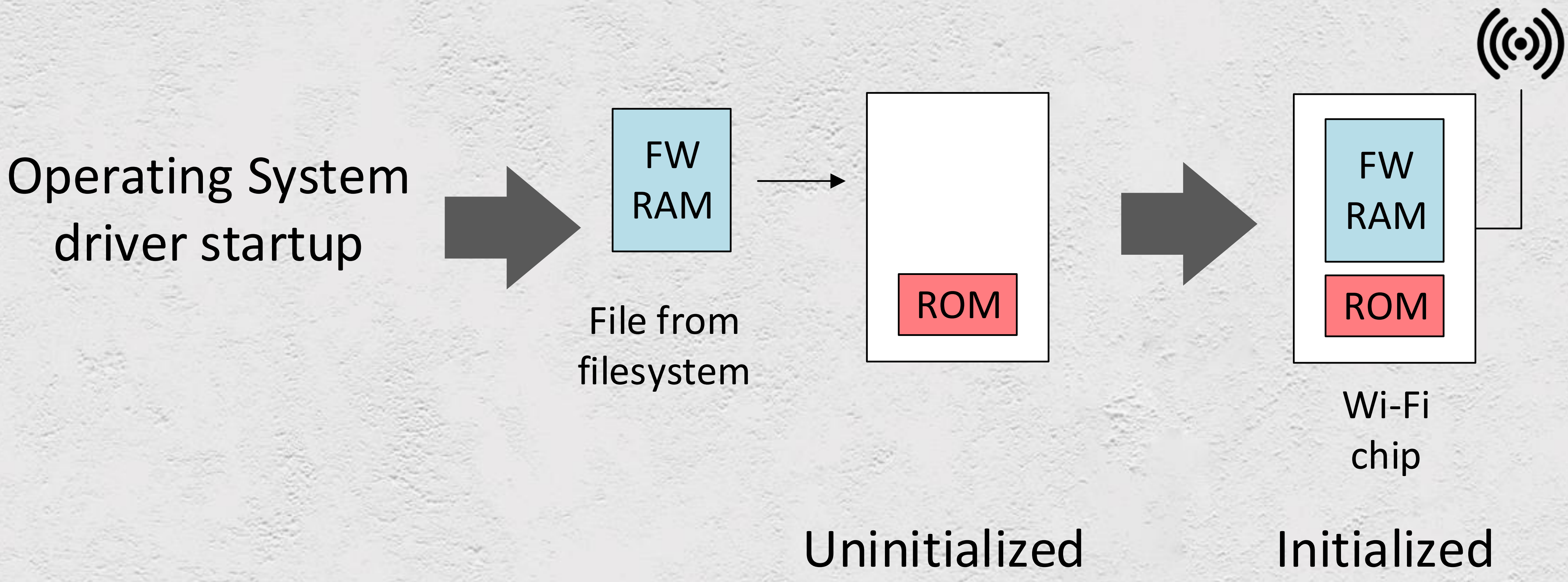
- Sony PlayStation 4, PlayStation 4 Pro
- Microsoft Surface, Surface Pro, Surface laptop, Xbox One
- Samsung Chromebook, some smartphones like Galaxy J1
- Valve SteamLink, and other devices…

How it works

- Difference between FullMAC and SoftMAC



How it starts up



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?

```
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;
        u8 data[1];
} __packed;
```

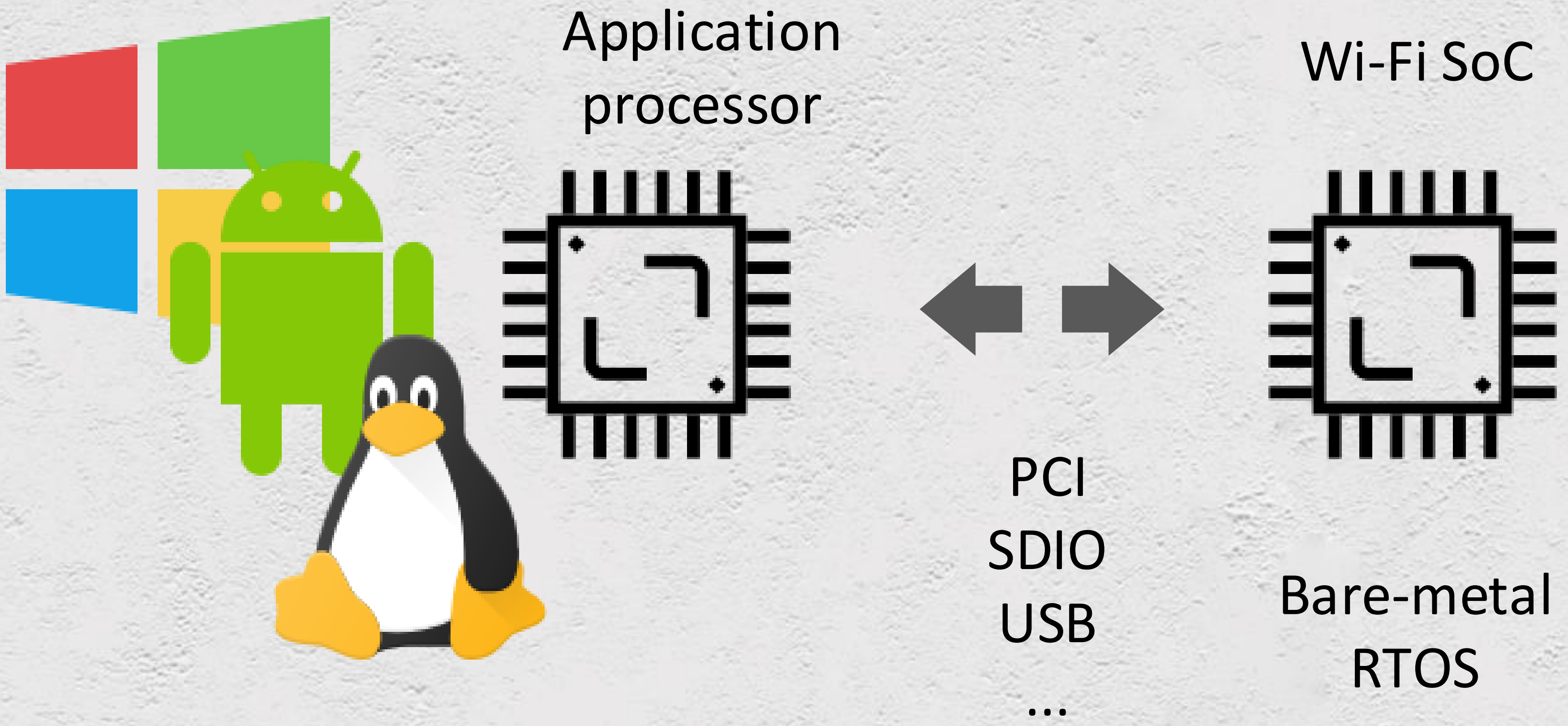
dnld_cmd | base_addr | data_length | crc

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	01	00	00	00	00	00	00	A0	00	02	00	00	0A	B6	BD	1B
00000010	1C	F0	9F	E5	1C	F0	9F	E5	1C	F0	9F	E5	1C	F0	9F	E5
00000020	1C	F0	9F	E5	1C	F0	9F	E5	1C	F0	9F	E5	1C	F0	9F	E5
00000030	00	28	0E	FF	64	22	00	00	AC	21	00	00	BC	21	00	00
00000040	14	22	00	00	24	22	00	00	34	22	00	00	44	22	00	00
00000050	50	22	00	00	00	00	00	EA	4F	07	00	EA	28	00	8F	E2
00000060	00	0C	90	E8	00	A0	8A	E0	00	B0	8B	E0	01	70	4A	E2
00000070	0B	00	5A	E1	48	07	00	0A	0F	00	BA	E8	14	E0	4F	E2
00000080	01	00	13	E3	03	F0	47	10	13	FF	2F	E1	28	48	03	00
00000090	68	48	03	00	00	30	A0	E3	00	40	A0	E3	00	50	A0	E3
000000A0	00	60	A0	E3	10	20	52	E2	78	00	A1	28	FC	FF	FF	8A
000000B0	82	2E	B0	E1	30	00	A1	28	00	30	81	45	1E	FF	2F	E1

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.

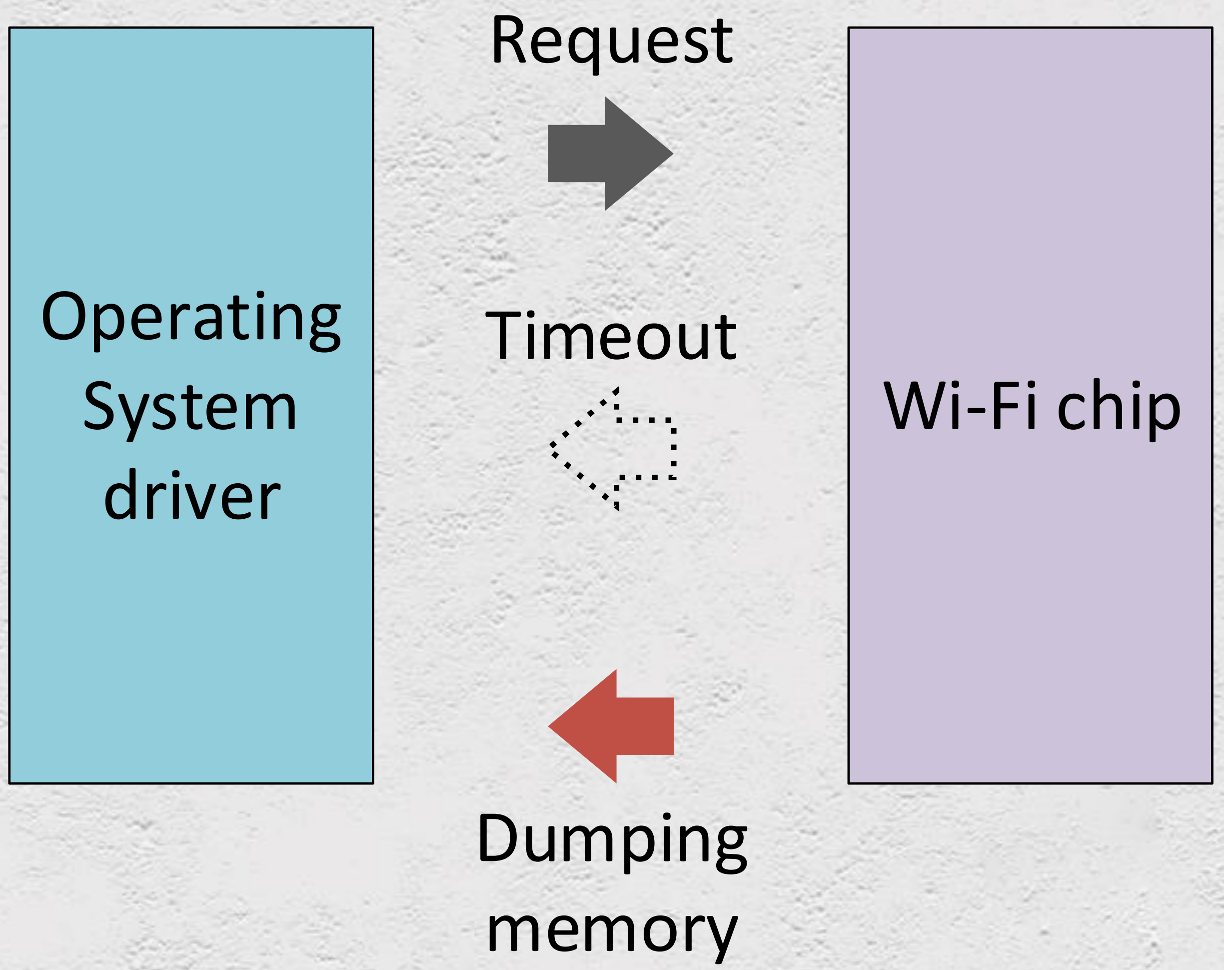


Firmware API implemented in driver

- 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



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

0x00000000 0x0000FFFF	ITCM
0x04000000 0x04007FFF	DTCM
0x80000000 0x90000000	UNKNOWN
0xC0000000 0xC00FFFFF	HEAP + THREAD STACK
0xFFD00000 0xFFFFFFFF	MAIN CODE + ROM

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)

```
typedef struct TX_THREAD_STRUCT
{
    /* The first section of the control block contains critical
       information that is referenced by the port-specific
       assembly language code.  Any changes in this section could
       necessitate changes in the assembly language.  */
    ULONG       tx_thread_id;           /* Control block ID         */
    ULONG       tx_run_count;           /* Thread's run counter     */
    VOID_PTR    tx_stack_ptr;           /* Thread's stack pointer   */
    VOID_PTR    tx_stack_start;         /* Stack starting address   */
    VOID_PTR    tx_stack_end;           /* Stack ending address     */
    ULONG       tx_stack_size;          /* Stack size               */
    ULONG       tx_time_slice;          /* Current time-slice       */
    ULONG       tx_new_time_slice;      /* New time-slice           */
```

```
/* Define thread control specific data definitions.  */

#define TX_THREAD_ID                    0x54485244UL
#define TX_THREAD_MAX_BYTE_VALUES       256
#define TX_THREAD_PRIORITY_MASK         0x1F
#define TX_THREAD_PRIORITY_GROUP_MASK   0xFF
#define TX_THREAD_GROUP_SIZE            8
#define TX_THREAD_GROUP_0               0
#define TX_THREAD_GROUP_1               8
#define TX_THREAD_GROUP_2               16
#define TX_THREAD_GROUP_3               24
```

ThreadX runtime objects in Steamlink Wi-Fi firmware

Object	Name	Entry point
Thread	Idle	0xFFD06479
Thread	MAC Tx	0xFFD50C39
Thread	MAC Tx Notify	0xFFD55B2F
Thread	MAC Mgmt	0xFFD13E55
Thread	CB Proc	0xFFD24859
Thread	IccTask	0xFFD066D5
Timer	SleepConfirmTmr	0xFFD1E055
Timer	AP_NullPktDoneTmr	0xFFD1DC55
Timer	NullPktDoneTmr	0xFFD1DC55
Queue	TxMgmt80211MsgQ	-
Queue	MacMgmtSMEMsgQ	-
Queue	TimerCbMsgQ	-

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

```
PUSH            {R0-R2,R4-R7,LR}
SUB             SP, SP, #0x28
MOVS            R6, R0
MOVS            R0, #0
LDR             R3, =off_4007D4C
STR             R0, [SP,#0x48+var_28]
STR             R0, [SP,#0x48+var_48]
LDR             R4, [R3] ; sub_FFFD01EA
MOVS            R7, R1
LDR             R2, [SP,#0x48+var_18]
MOVS            R0, R6
ADD             R3, SP, #0x48+var_28
BLX             R4      ; sub_FFFD01EA
CMP             R0, #0
BNE             loc_FFFE42B8
LDR             R4, [R6,#8]
MOVS            R5, #0
B               loc_FFFE42AC
```

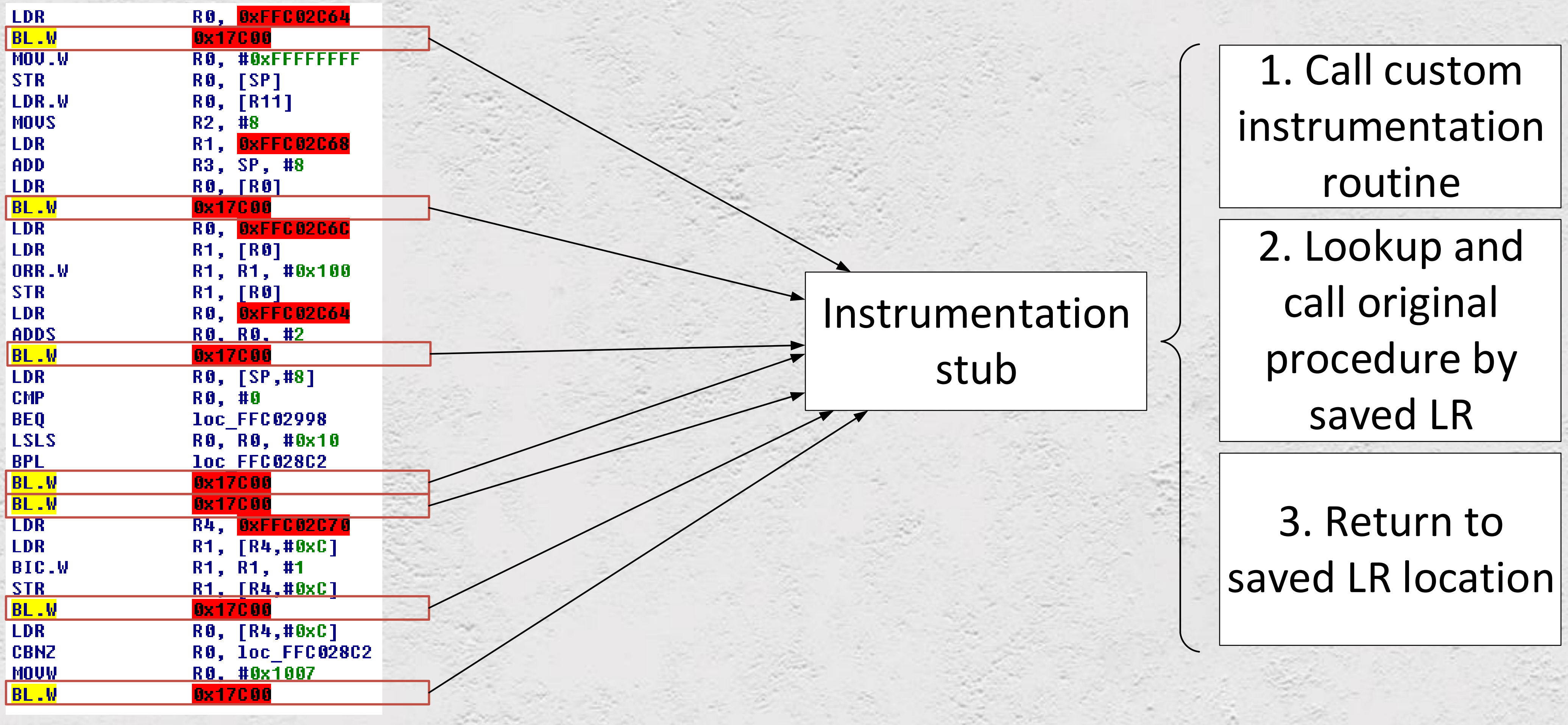
Firmware instrumentation

- Detour all calls in memory region to the instrumentation tool

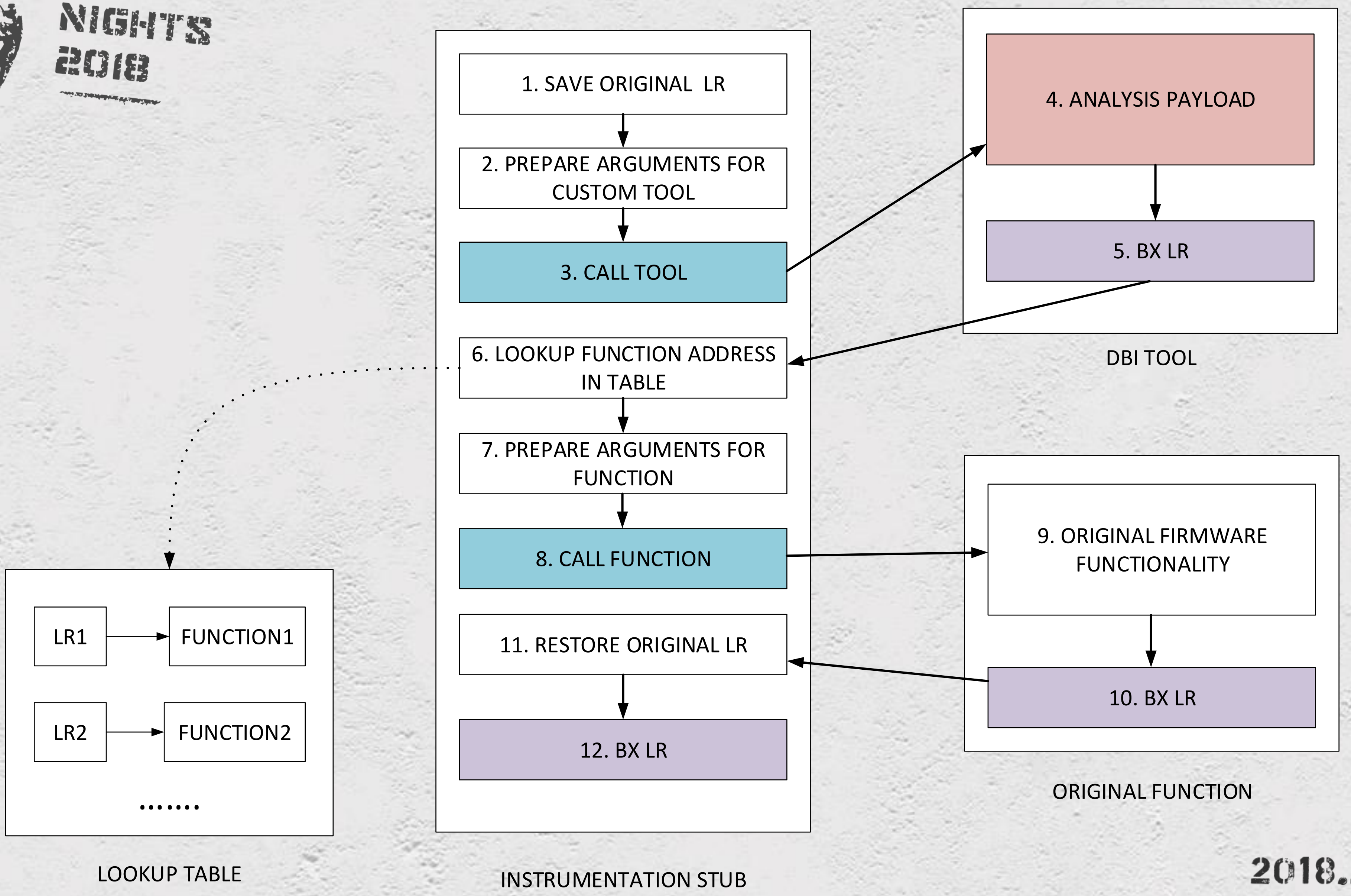
```
LDR      R0, =0x30200
BL       sub_FFC2BCCC
MOV.W    R0, #0xFFFFFFFF
STR      R0, [SP,#0x20+var_20]
LDR.W    R0, [R11] ; dword_FF8005D4
MOVS     R2, #8
LDR      R1, =0xCBFFFF
ADD      R3, SP, #0x20+var_18
LDR      R0, [R0]
BL       sub_1B74
LDR      R0, =0x80003434
LDR      R1, [R0]
ORR.W    R1, R1, #0x100
STR      R1, [R0]
LDR      R0, =0x30200
ADDS     R0, R0, #2
BL       sub_FFC2BCCC
LDR      R0, [SP,#0x20+var_18]
CMP      R0, #0
BEQ      loc_FFC02998
LSLS     R0, R0, #0x10
BPL      loc_FFC028C2
BL       sub_FFC15638
BL       j_j_TX_DISABLE_INTERRUPTS
LDR      R4, =byte_FF8005C4
LDR      R1, [R4,#(dword_FF8005D0 - 0xFF8005C4)]
BIC.W    R1, R1, #1
STR      R1, [R4,#(dword_FF8005D0 - 0xFF8005C4)]
BL       disable_or_restore_interrupts
LDR      R0, [R4,#(dword_FF8005D0 - 0xFF8005C4)]
CBNZ     R0, loc_FFC028C2
MOVW     R0, #0x1007
BL.W     sub_8AF8
```

```
LDR      R0, 0xFFC02C64
BL.W     0x17C00
MOV.W    R0, #0xFFFFFFFF
STR      R0, [SP]
LDR.W    R0, [R11]
MOVS     R2, #8
LDR      R1, 0xFFC02C68
ADD      R3, SP, #8
LDR      R0, [R0]
BL.W     0x17C00
LDR      R0, 0xFFC02C6C
LDR      R1, [R0]
ORR.W    R1, R1, #0x100
STR      R1, [R0]
LDR      R0, 0xFFC02C64
ADDS     R0, R0, #2
BL.W     0x17C00
LDR      R0, [SP,#8]
CMP      R0, #0
BEQ      loc_FFC02998
LSLS     R0, R0, #0x10
BPL      loc_FFC028C2
BL.W     0x17C00
BL.W     0x17C00
LDR      R4, 0xFFC02C70
LDR      R1, [R4,#0xC]
BIC.W    R1, R1, #1
STR      R1, [R4,#0xC]
BL.W     0x17C00
LDR      R0, [R4,#0xC]
CBNZ     R0, loc_FFC028C2
MOVW     R0, #0x1007
BL.W     0x17C00
```

Firmware instrumentation cont.



Instrumentation stub



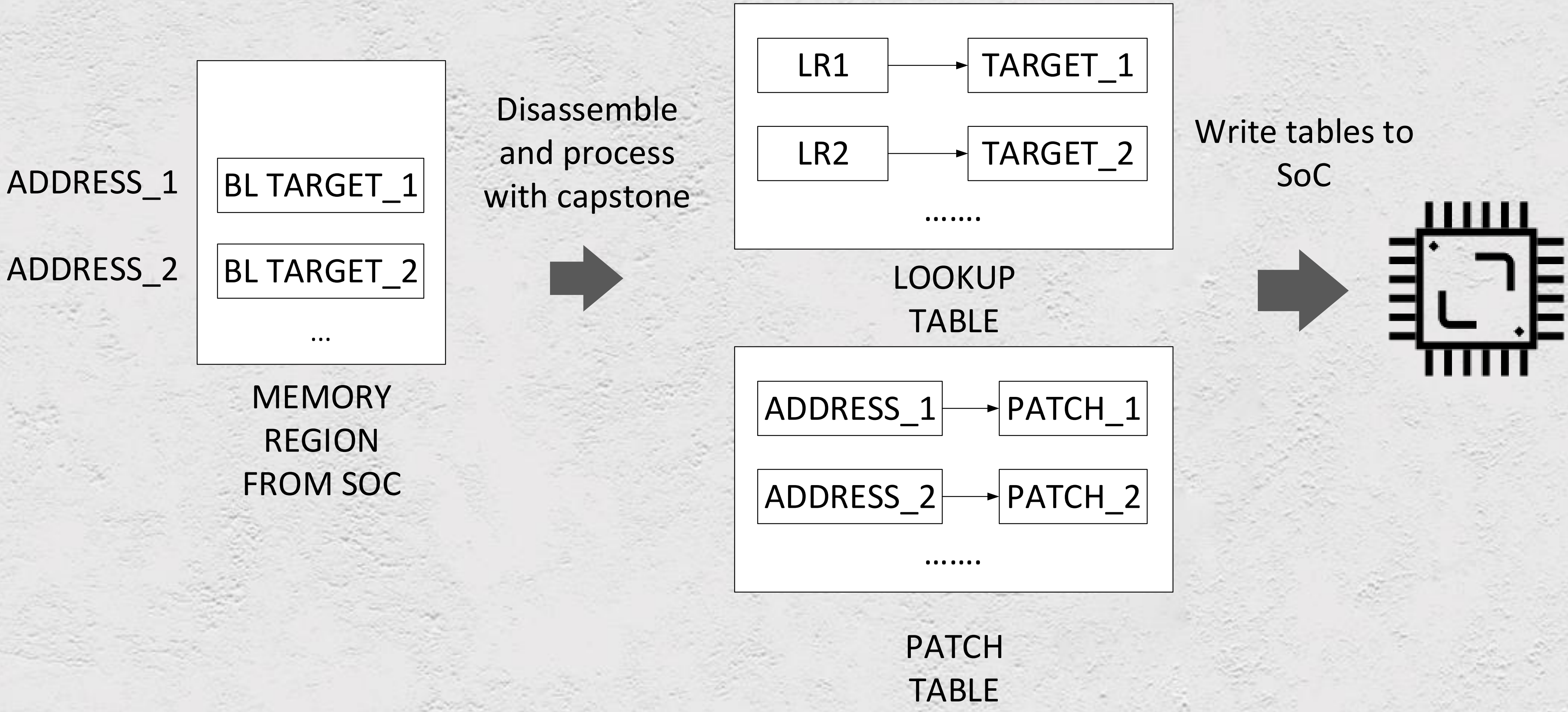
How to achieve this?

- 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 (4bytes)
  - 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

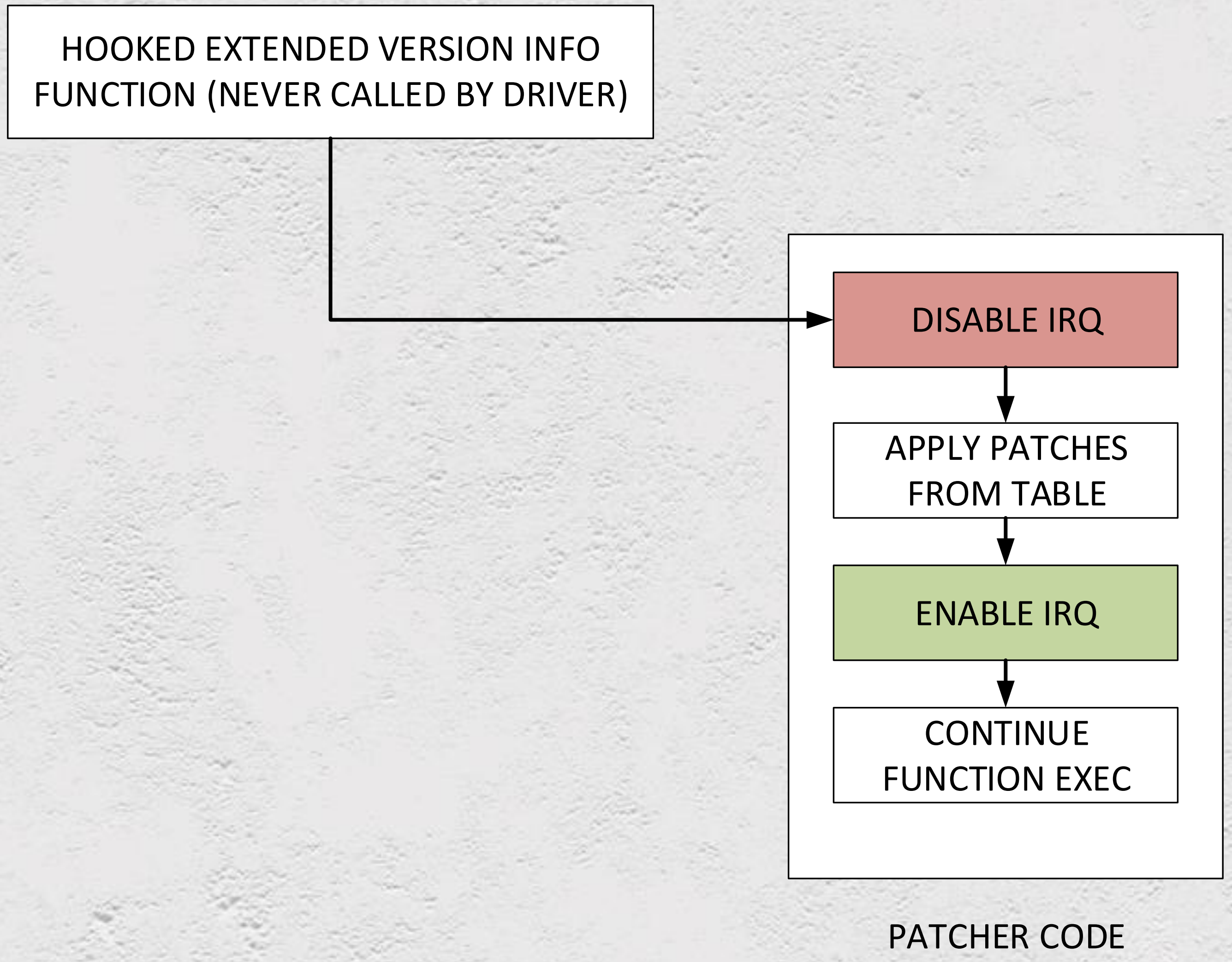
Firmware instrumentation. Code that runs on AP. cont



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



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

```
                american fuzzy lop 2.52b (fuzzer)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 4 days, 7 hrs, 27 min, 54 sec      │  cycles done : 2526  │
│   last new path : 4 days, 4 hrs, 12 min, 13 sec      │  total paths : 128   │
│ last uniq crash : 4 days, 6 hrs, 40 min, 18 sec      │ uniq crashes : 11    │
│  last uniq hang : 4 days, 2 hrs, 41 min, 44 sec      │   uniq hangs : 15    │
├─ cycle progress ────────────────┬─ map coverage ─────┴──────────────────────┤
│  now processing : 96 (75.00%)   │    map density : 0.44% / 2.59%           │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.08 bits/tuple          │
├─ stage progress ────────────────┼─ findings in depth ───────────────────────┤
│  now trying : havoc             │ favored paths : 76 (59.38%)               │
│ stage execs : 430/573 (75.04%)  │  new edges on : 86 (67.19%)               │
│ total execs : 203M              │ total crashes : 4.89M (11 unique)         │
│  exec speed : 544.4/sec         │  total tmouts : 40.3k (21 unique)         │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ──────────┤
│   bit flips : 14/98.6k, 9/98.4k, 5/98.2k        │    levels : 9            │
│  byte flips : 2/12.3k, 0/6584, 0/6425           │   pending : 0            │
│ arithmetics : 16/372k, 0/343k, 0/37.8k          │  pend fav : 0            │
│  known ints : 0/23.7k, 1/163k, 2/267k           │ own finds : 118          │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a          │
│       havoc : 79/74.2M, 1/127M                  │ stability : 100.00%      │
│        trim : 29.56%/5416, 44.78%               ├──────────────────────────┘
└─────────────────────────────────────────────────┘          [cpu000: 29%]
```

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

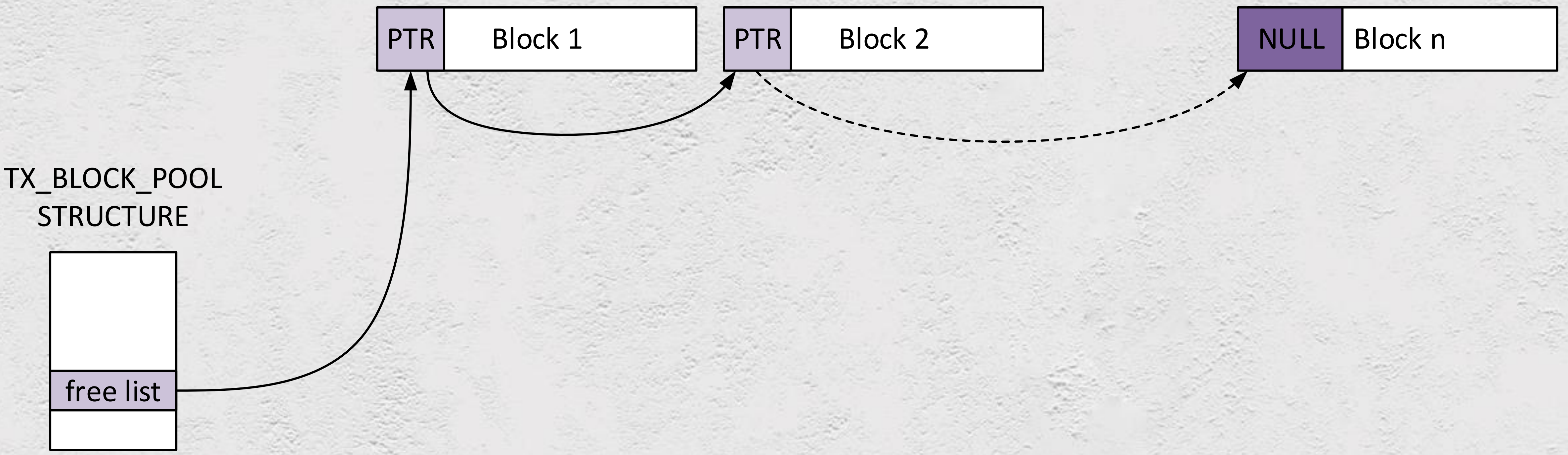
Testing on other devices

- 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)

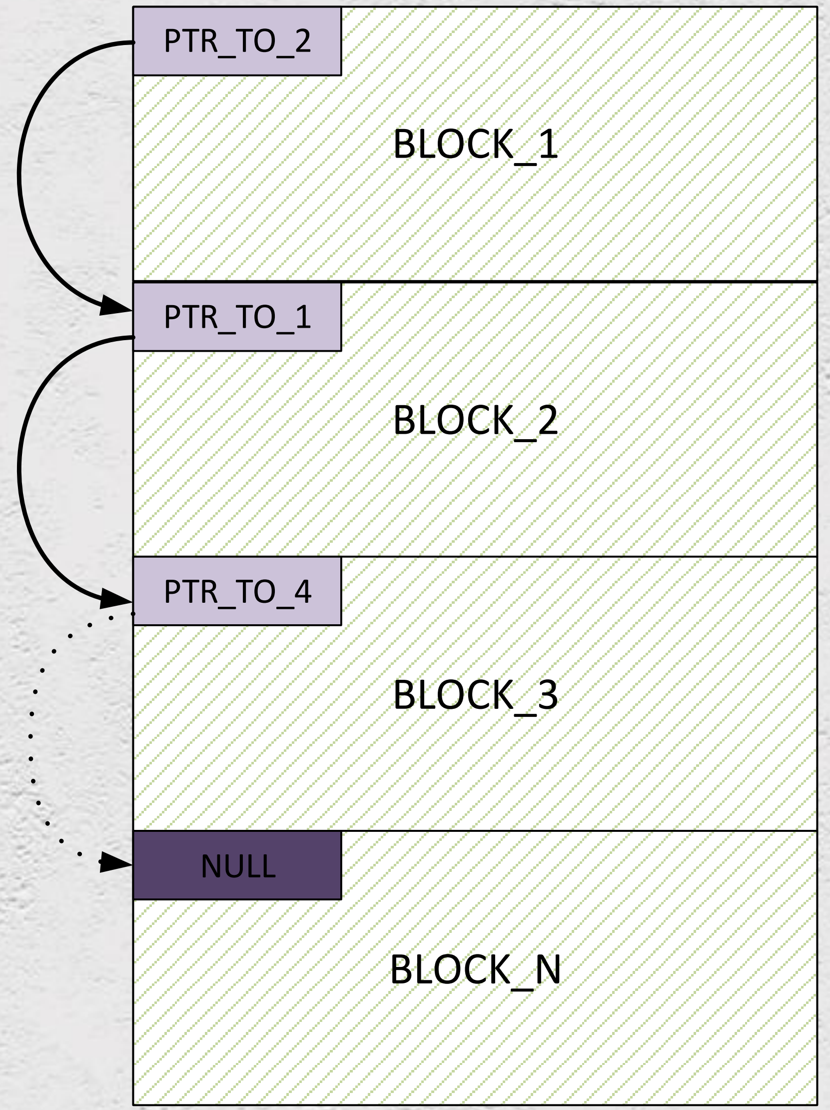
The most interesting bug to be exploited

- 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

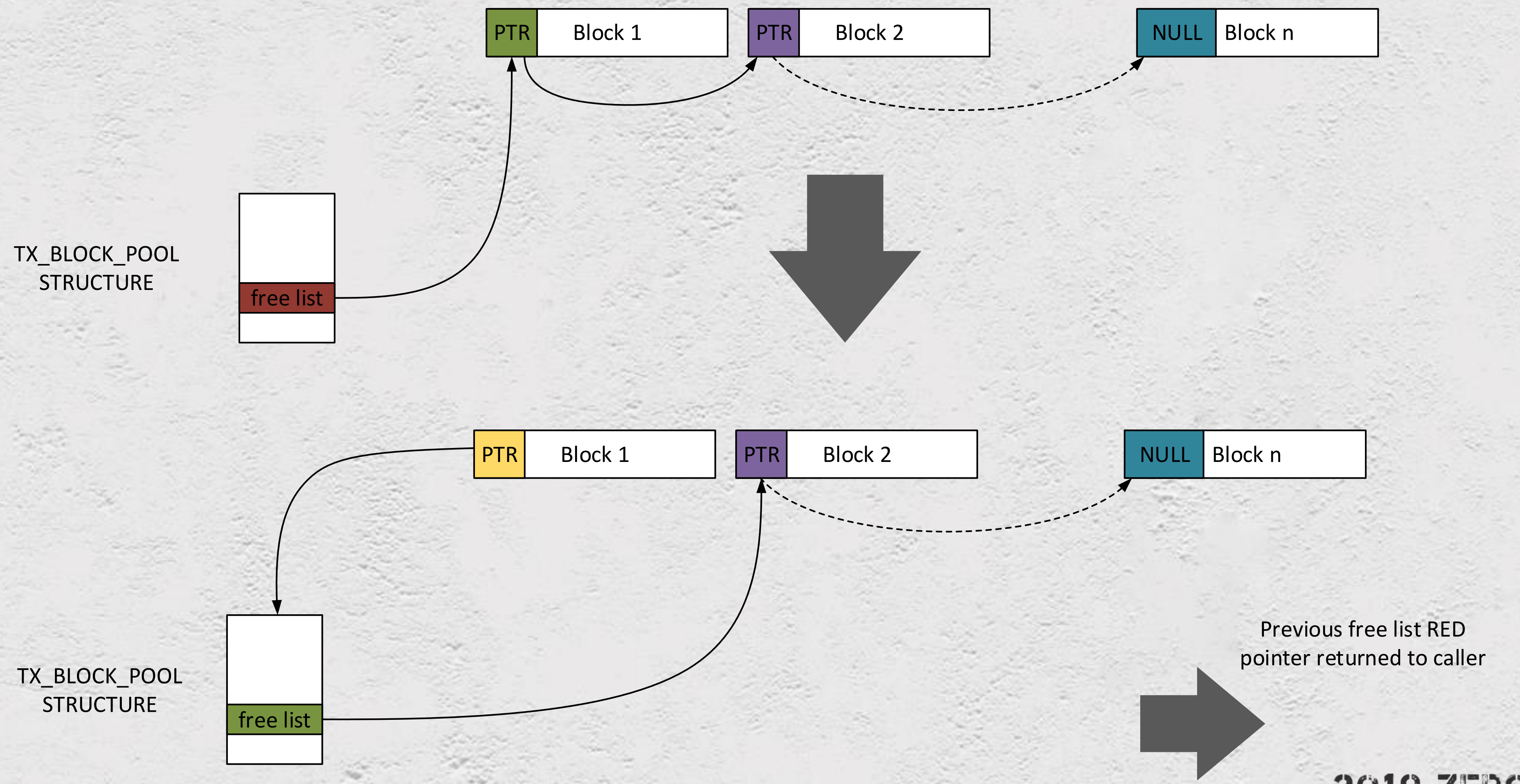


ThreadX block pool overview cont.



LINKS CAN BE REARANGED

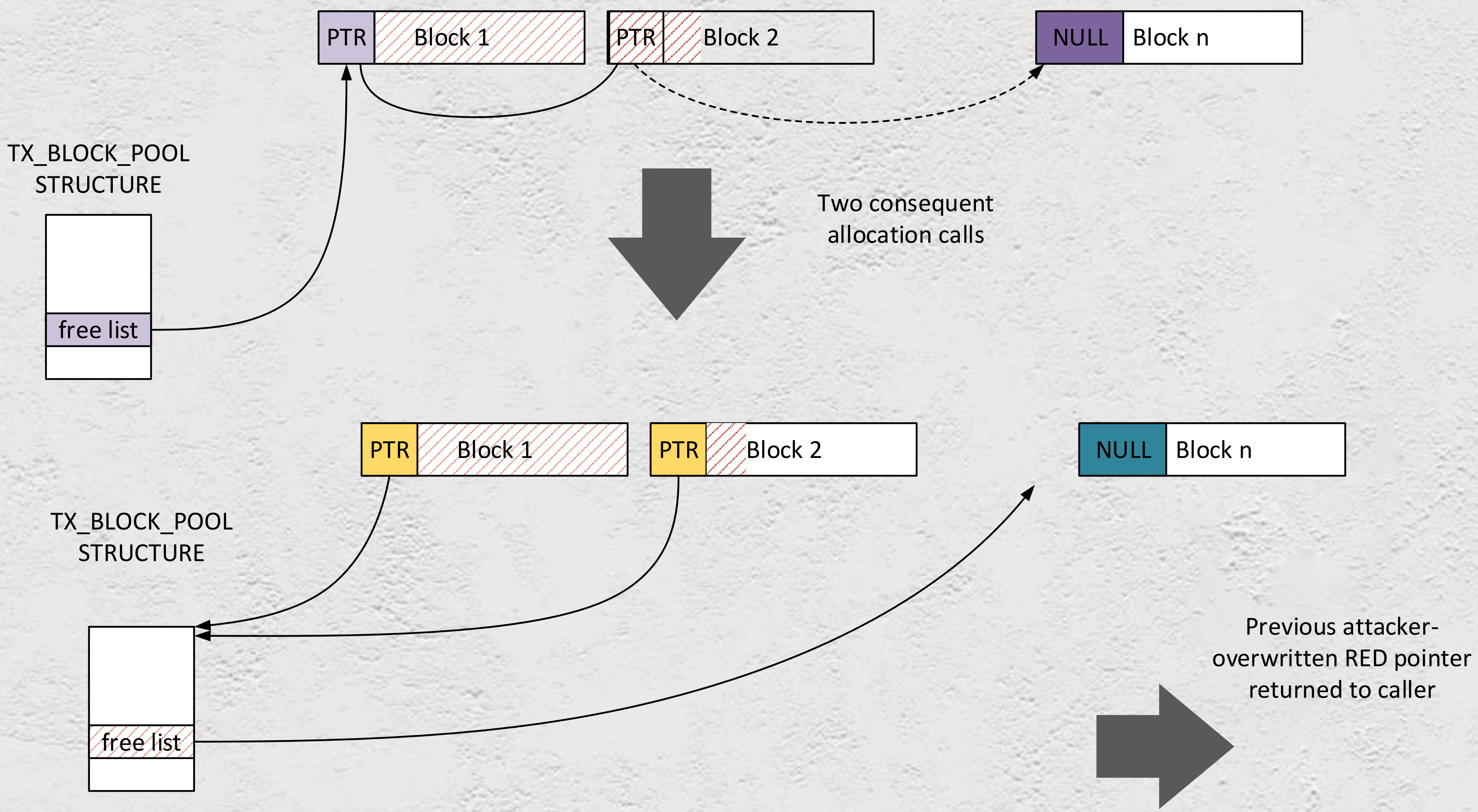
ThreadX block pool allocation of block



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

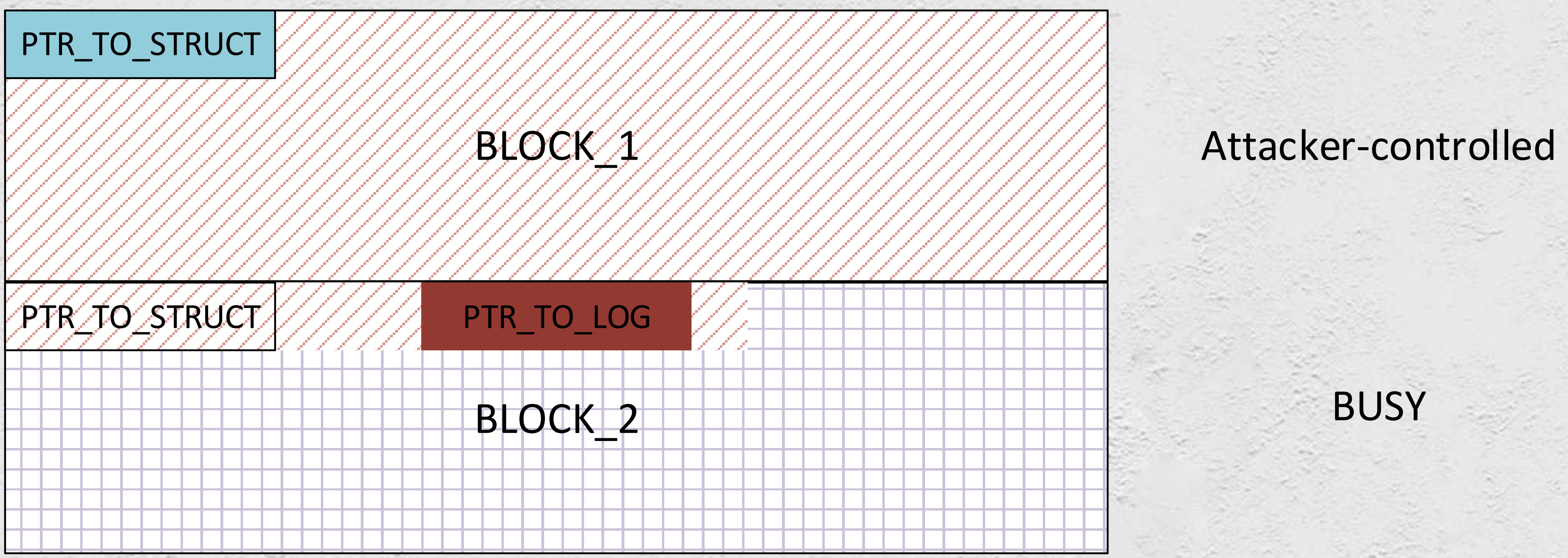


Exploitation – a simpler way

- 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

```
if ( *(unsigned __int8 *)(memory_ptr + 0xD) << 31 )
{
  pre_release_callback = *(int (__fastcall **)(int))(memory_ptr + 0x18);
  if ( pre_release_callback )
    post_release_callback = (void (*)(void))pre_release_callback(memory_ptr);
}
result = tx_block_release(memory_ptr);
if ( post_release_callback )
  post_release_callback();
if ( result )
  return 1;
return 0;
```

Exploitation – a simpler way



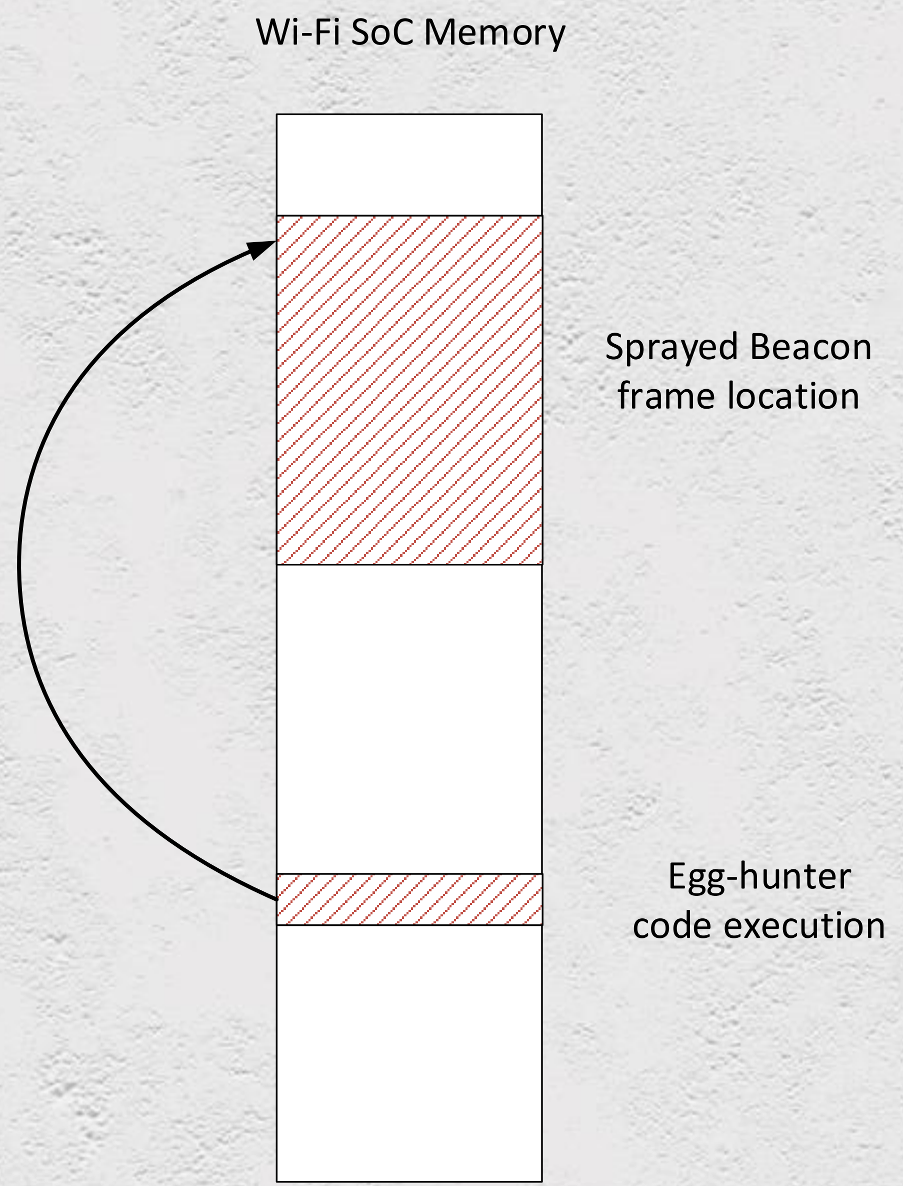
Exploiting Valve Steamlink

- 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)

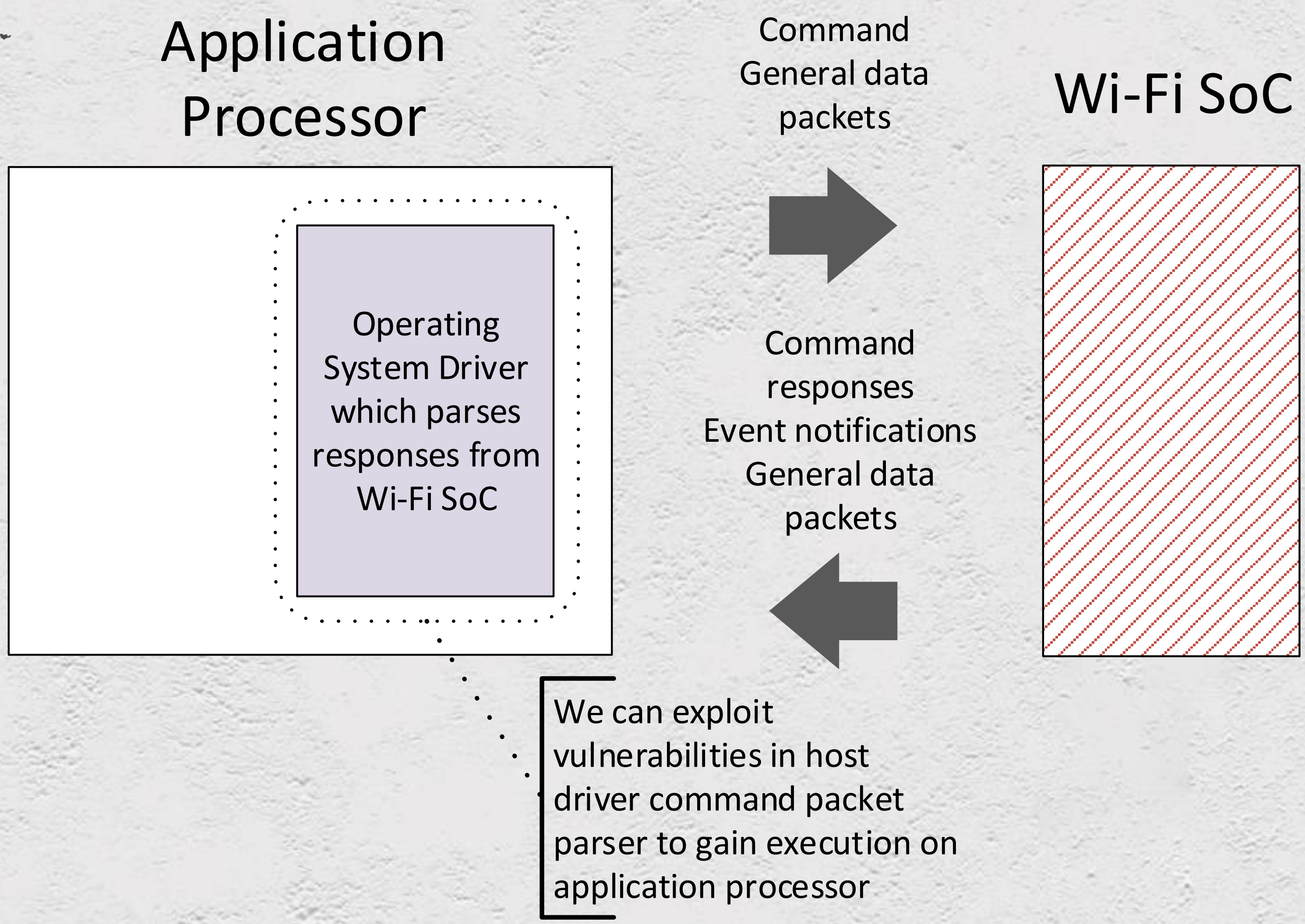
Exploit – stage1

- 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

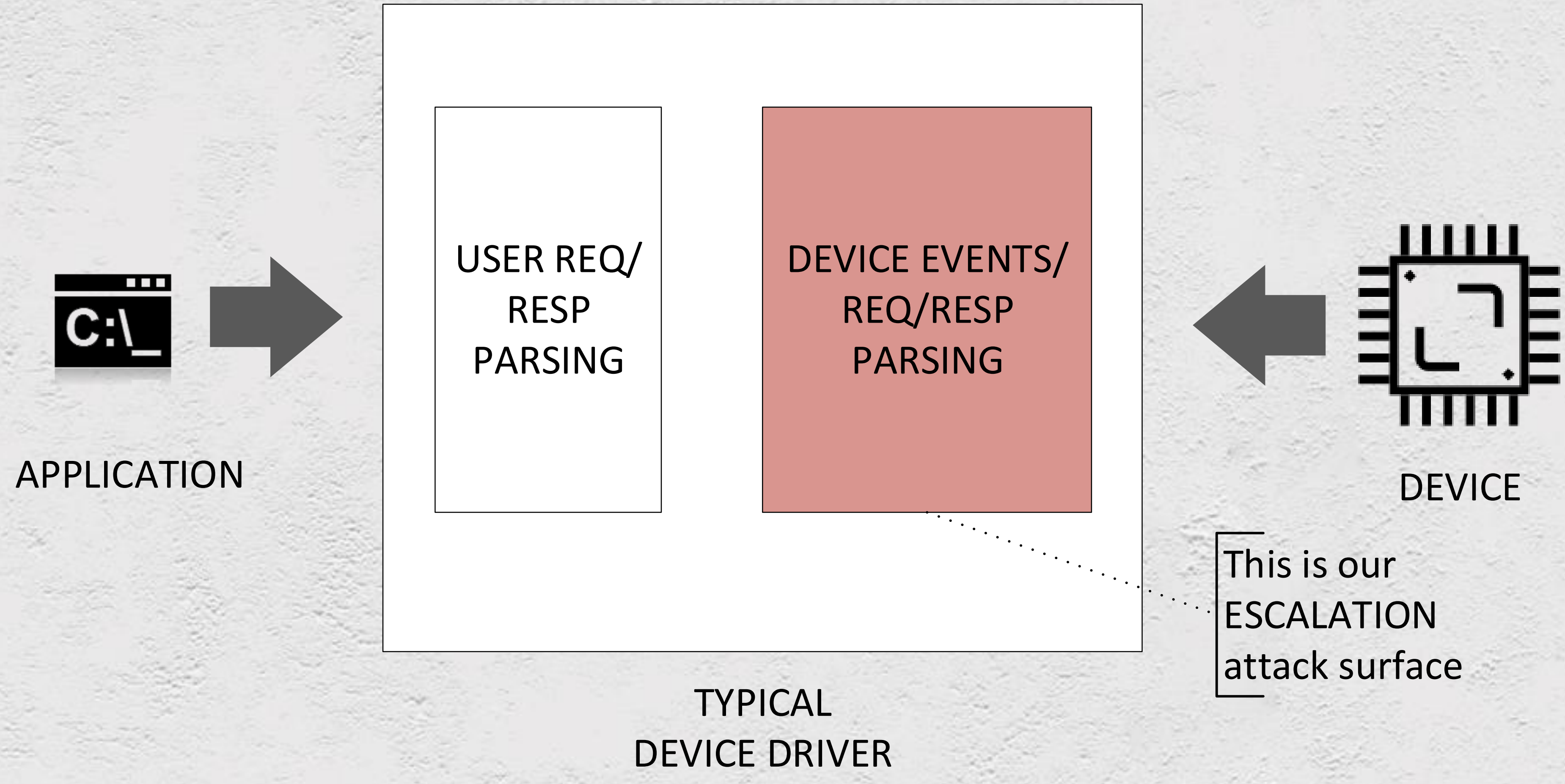
Exploit – stage1 pic



Escalation attack surface



Escalation attack surface cont.



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

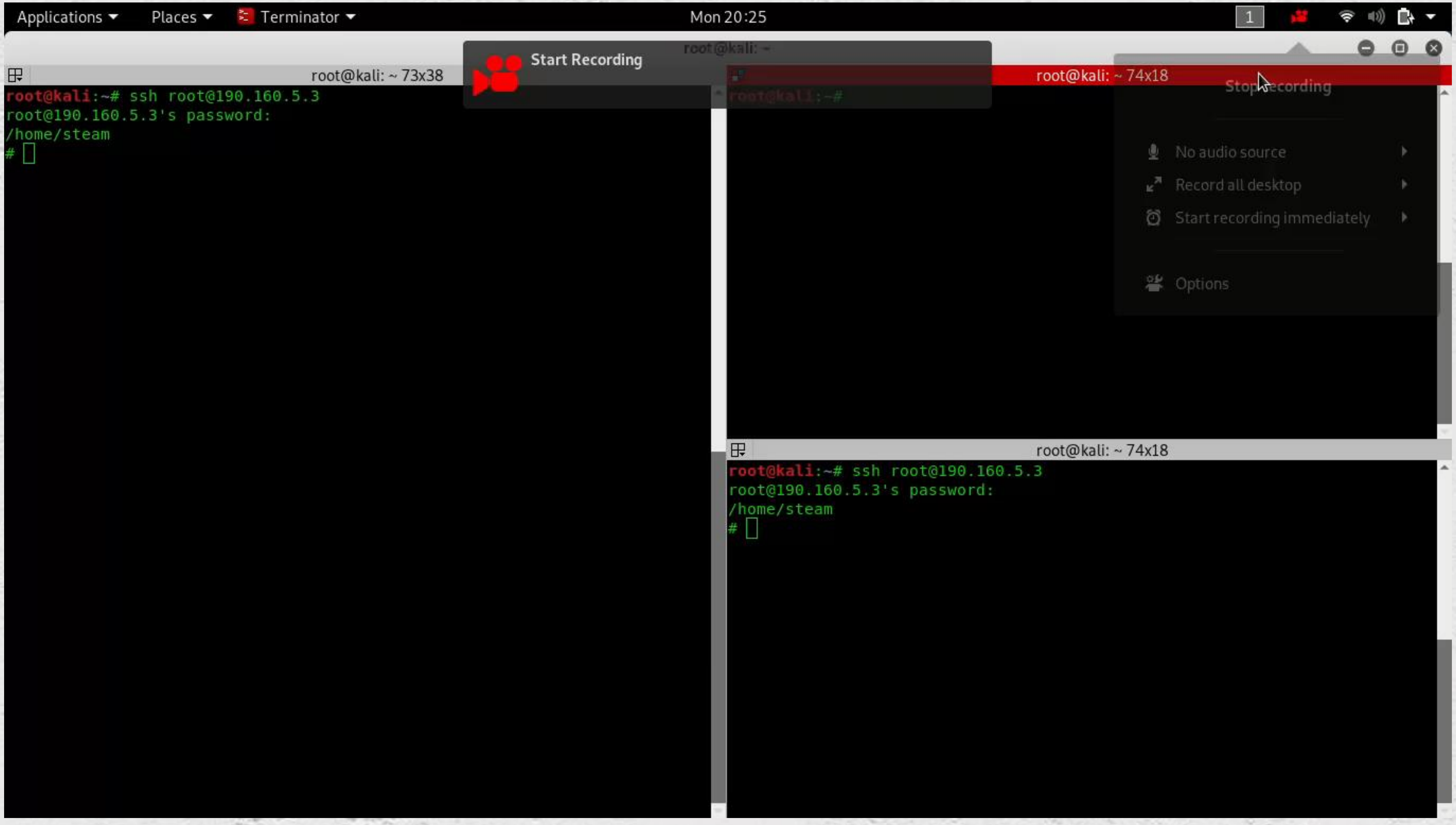
Exploit – stage3

- 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

Demo


Applications
Places
Terminator
Mon 20:25
Start Recording
root@kali: ~ 73x38
root@kali:~# ssh root@190.160.5.3
root@190.160.5.3's password:
/home/steam
#
root@kali: ~ 74x18
No audio source
Record all desktop
Start recording immediately
Options
root@kali: ~ 74x18
root@kali:~# ssh root@190.160.5.3
root@190.160.5.3's password:
/home/steam
#

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