2020-01-02

Exploiting Wi-Fi Stack on Tesla Model S

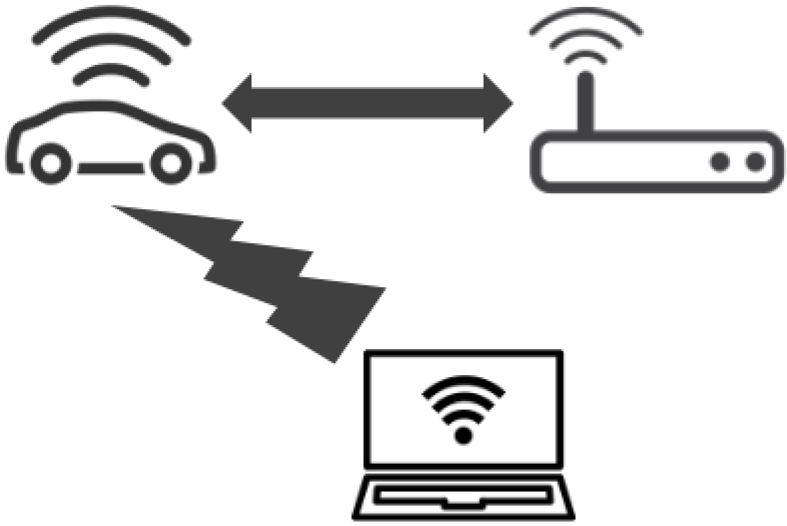
by Tencent Keen Security Lab

In the past two years, Keen Security Lab did in-depth research on the security of Tesla Cars and presented our research results on Black Hat 2017 and Black Hat 2018. Our research involves many in-vehicle components. We demonstrated how to hack into these components, including CID, IC, GATEWAY, and APE. The vulnerabilities we utilized exists in the kernel, browser, MCU firmware, UDS protocol, and OTA updating services. It is worth noting that recently we did some interesting works on Autopilot module, we analyzed the implementation details of autowipers and lane recognition function and make an example of attacking in the physical world.

To understand the security of Tesla\'s on-board system more comprehensively, we researched the Wi-Fi module (aka Parrot on Model S) and found two vulnerabilities in the Wi-Fi firmware and Wi-Fi driver. By combining these two vulnerabilities, the host Linux system can be compromised.

Introduction

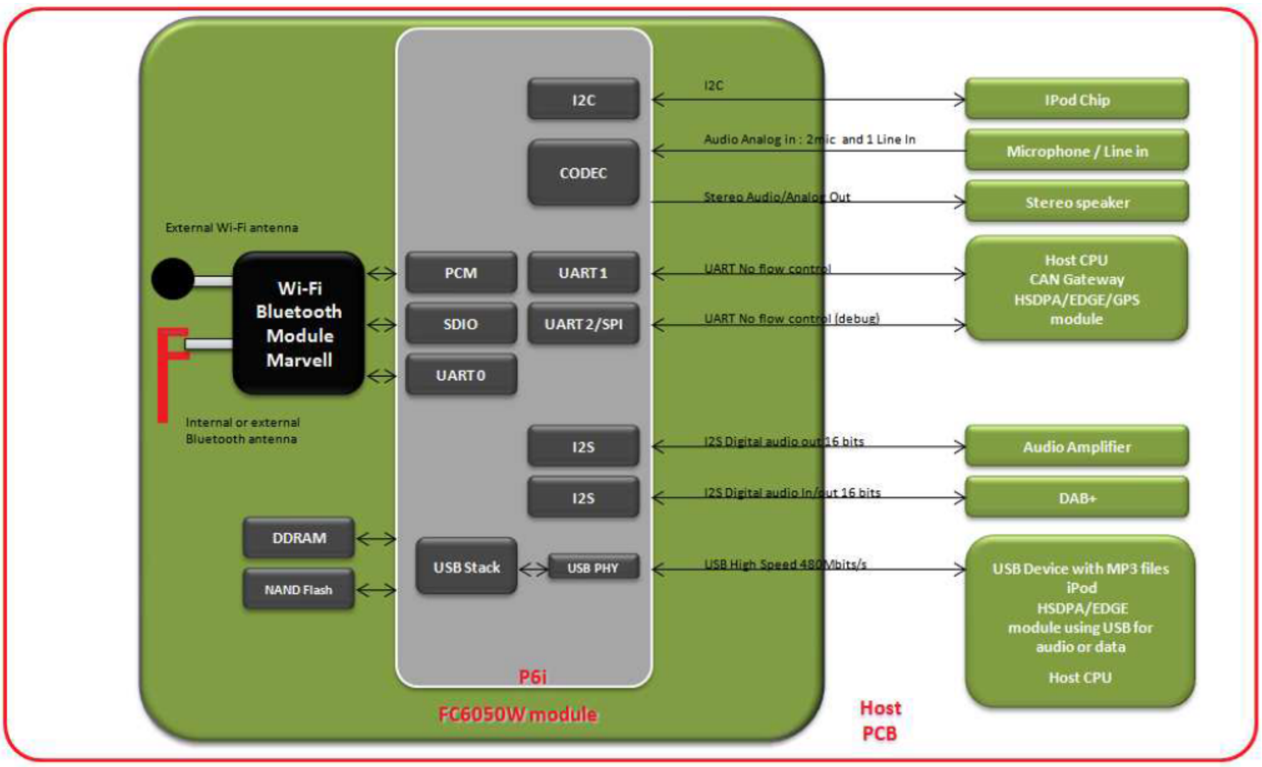
This article reveals the details of two vulnerabilities and introduces how to exploit these vulnerabilities, which proves that these vulnerabilities can be used by an attacker to hack into the Tesla Model S in-vehicle system remotely through the Wi-Fi.



Parrot Module

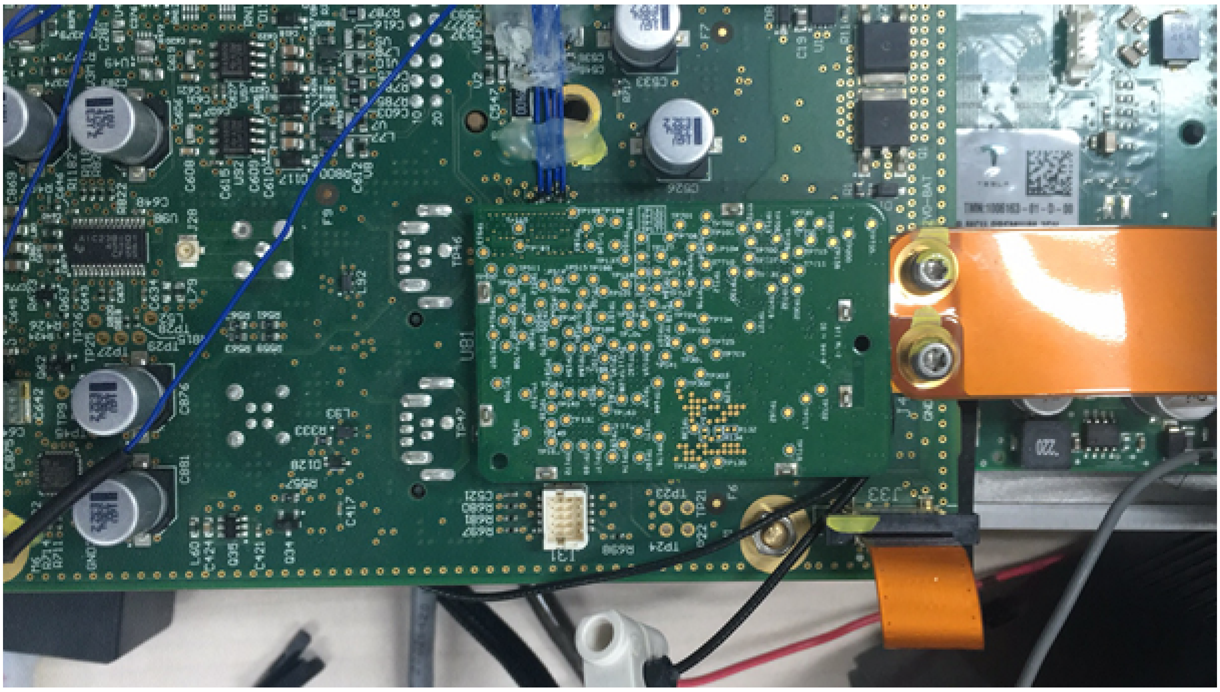
The third-party module Parrot on Tesla Model S is FC6050W, which integrates the Wireless function and Bluetooth function. Parrot connects to CID via USB protocol and runs Linux. Parrot uses the USB Ethernet gadget so that Parrot can communicate with CID trough Ethernet. When Tesla Model S connected to a wireless network, it is Parrot connected to the wireless network. Then, the network traffic from CID routed by Parrot.

We can find the hardware organization from a very detailed datasheet[1].



The pinout description of Parrot also presented in the datasheet. The Linux shell can be found through the Debug UART pins.

PIN	FUNCTION	PIN TYPE	COMMENT
2	RESET_IPOD	O	IPOD Reset
4	I2C_SCL	O	I2C Clock
6	I2C_SDA	I/O	I2C Data
8	UART_AT_TX	O	AT Commands & flash update UART output
10	UART_AT_RX	I	AT Commands & flash update UART input
12	UART_DBG_TX	O	Debug UART output
14	UART_DBG_RX	I	Debug UART input
16	**VCC**	P	**Power supply : 3.3v**
18	I2S_IN1	I	Digital audio data input 1
20	I2S_OUT2	O	Digital audio data output 2
22	I2S_OUT1	O	Digital audio data output 2
24	I2S_FSYNC	O	Digital audio frame synchronization
26	I2S_CLK	O	Digital audio clock
28	I2S_MCLK	O	Digital audio master clock
30	BOOTS	I	Discrete boots mode signal – active high
32	LINE_OUT_R	O	Analog audio output - Right
34	**GND**	P	**Ground**
36	MIC_2_P	I	Positive microphone 2 input
38	MIC_2_N	I	Negative microphone 2 input
40	LINE_IN_R	I	Analog audio input - Right



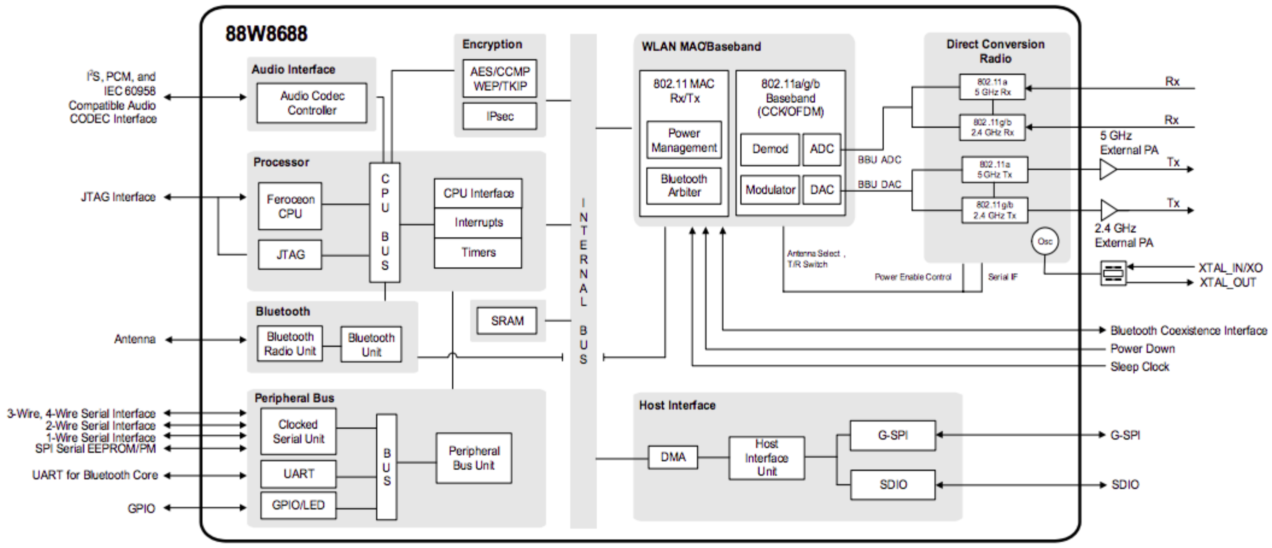
The reset pin connects to the GPIO port of CID. Thus CID can reset the whole Parrot module by using these commands.

```
echo 1 \> /sys/class/gpio/gpio171/value
sleep 1
echo 0 \> /sys/class/gpio/gpio171/value
```

Marvell Wifi Chip

The Marvell 88W8688 is a low-cost, low-power highly-integrated IEEE 802.11a/g/b MAC/Baseband/RF WLAN and Bluetooth Baseband/RF system-on-chip (SoC) [2].

The block diagram published on the Marvell website[3].



The 88w8688 contains an embedded high-performance Marvell Ferocean ARM9-compatible processor. By modifying the firmware, we acquired the value of the Main ID Register, which is 0x11101556. According to the value, we concluded the CPU might be Feroceon 88FR101 rev 1. On Parrot, the Marvell 88w8688 chipset connects to the host system via the SDIO interface.

The memory region of 88w8688 could be as follows.

0x00000000-0x0005FFFF	ITCM	Firmware Code
0x03F00000-0x03F7FFFF	ROM	Bootloader, ThreadX RTOS, Bluetooth Code
0x04000000-0x0400FFFF	DTCM	Heap, Stack
0x80000000-0x8000FFFF	IO MEMORY	SDIO Interface Register, …
0x90000000-0x9000FFFF	IO MEMORY	UART Register, …
0xC0000000-0xC003FFFF	RAM	Heap, Global variables, …

Firmware

The firmware download process of 88w8688 contains two stages, the helper firmware “sd8688_helper.bin” downloads to chip first, then the main firmware “sd8688.bin” downloads to chip. The helper responsible for and downloading the firmware file and verifying every chunk of the firmware file. The firmware file consists of many chunks, below is the structure of each chunk stable.

```
struct fw_chunk {
  int chunk_type;
  int addr;
```

```
    unsigned int length;
    unsigned int crc32;
    unsigned char [1];
} __packed;
```

The 88w8688 chip runs based on ThreadX OS which is an RTOS targeting for embedded devices. The code of ThreadX can be found in the ROM region, so the firmware “sd8688.bin” runs as an application of ThreadX.

On Tesla, the version ID of firmware “sd8688.bin” is “sd8688-B1, RF868X, FP44, 13.44.1.p49”. All the following research results are based on this version.

After identified the ThreadX API, the information about tasks is as below.

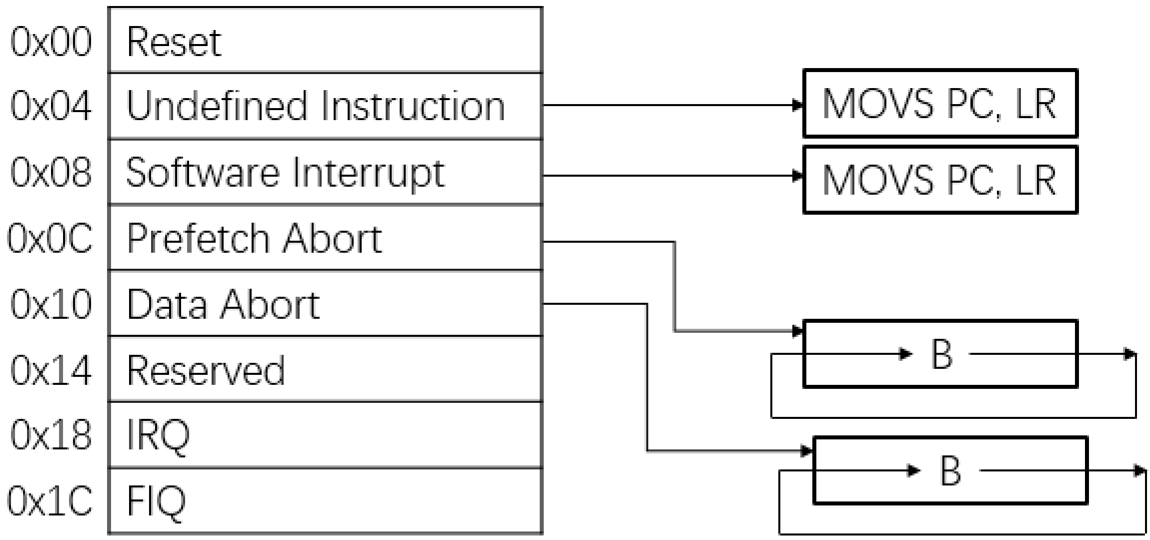
Task Name	Entry Function	Stack Region
Idle	0x0000F970	C0025AB0-C0025C7C
MAC Mgmt	0x0001202C	C00183FC-C0018848
MAC Tx Notify	0x0123E0	0x400B2F4-0x400B6B8
AmphciTask	0x028B68	0x04008CC8-0x04008EC8
CB Proc	0x002E4F4	0x0400A4A4-0x400A8A4
MAC Tx	0x0329A4	0x0400A8A4-0x0400B0A4

Also, the information about memory pools is as below.

Start Address	Block Size	Block Number	Total Size	Memory Pool Name
C000DC00	0x7FC	4	0x2000	pool_start_id_tx
C000FC00	0x7FC	4	0x2000	pool_start_id_rx
C0011C00	0x3FC	4	0x1000	pool_start_id_sme
C0012C00	0x1FC	5	0xA00	pool_start_id_mlme
C0013600	0x9FC	1	0x800	pool_start_id_scanbuf
C0014000	0x7FC	3	0x1800	pool_start_id_cmdbuf
C0015800	0x1FC	6	0xC00	pool_start_id_evtbuf
C0016400	0x7FC	1	0x800	pool_start_id_rmlmebuf
C0016C00	0x3FC	2	0x800	pool_start_id_encrbuf

Log and Debug

The firmware did not implement the CPU vector handler for Data Abort, Prefetch Abort, Undefine, and SWI, which means the firmware halts after a crash, and we cannot know where and why the firmware crash.



So, we patched the firmware with our custom Prefetch Abort and Data Abort vector handler. The handler records the values of register includes general-purpose register, the status register, and link register in system mode and IRQ mode. In this way, we can know where the code runs in both system mode and IRQ mode when a crash happens.

We chose to write these values to unused memory, for example, 0x52100~0x5FFFF. These values still can be read after the chip reset.

After implemented the undefine vector handler and changed some instruction to undefine instruction, we can get or set registers when the firmware is running. In this way, we can debug the firmware.

To re-download a new firmware to chip, try to send the command HostCmd_CMD_SOFT_RESET from kernel to chip, then the chip resets and new firmware downloads.

Vulnerability in Firmware

The 88w8688 chip supports 802.11e WMM (Wi-Fi Multimedia) protocol. In this protocol, the station could send an action frame Add Traffic Stream (ADDTS) request with Traffic Specification (TSPEC) to another device. Then the other device returns an action frame ADDTS response. Below is the action frame.

Action Category	Action Code	Dialog Token	Status Code	TSPEC	...
0x11	0x00:Req 0x01:Resp	0x00	0x00	...	...

The whole process of ADDTS may like this. When the host operation system wants to send an ADDTS request, the kernel driver fills and sends a HostCmd_DS_COMMAND structure with command HostCmd_CMD_WMM_ADDTS_REQ to chip. Then the firmware transmits the ADDTS request packet over the air. When the chip received an ADDTS response from another device, it copies this response without an action header to the HostCmd_CMD_WMM_ADDTS_REQ structure as a result of ADDTS_REQ command and passes the structure HostCmd_DS_COMMAND to the kernel driver. After that, the kernel driver process this response.

```
struct _HostCmd_DS_COMMAND
{
    u16 Command;
    u16 Size;
    u16 SeqNum;
    u16 Result;
    union
    {
        HostCmd_DS_GET_HW_SPEC hwspec;
        HostCmd_CMD_WMM_ADDTS_REQ;
        //…….
     }
}
```

The vulnerability exists in the process of copying the data from the ADDTS response packet to the HostCmd_CMD_WMM_ADDTS_REQ structure. The length of copy calculated by subtracting 4 bytes length of action header from length of action frame. But if the action frame only contains a header and the length of the header is only 3 bytes, the length needs to copy is 0xffffffff. So, the memory could be corrupted very badly, resulting in a crash very stable.

Vulnerability in Driver

There are three kinds of data sent between the chip and the kernel driver through the SDIO interface, MV_TYPE_DATA, MV_TYPE_CMD, and MV_TYPE_EVENT. The definition of commands and events can be found in source code.

```
/** Host Command option for wait for RSP */
#define HostCmd_OPTION_WAITFORRSP            0x0002
/** Host Command option for wait for RSP Timeout */
#define HostCmd_OPTION_TIMEOUT               0x0004

/** Host Command ID : Get hardware specifications */
#define HostCmd_CMD_GET_HW_SPEC              0x0003
/** Host Command ID : 802.11 scan */
#define HostCmd_CMD_802_11_SCAN              0x0006
/** Host Command ID : 802.11 get log */
#define HostCmd_CMD_802_11_GET_LOG           0x000b
/** Host Command ID : MAC multicast address */
#define HostCmd_CMD_MAC_MULTICAST_ADR        0x0010
/** Host Command ID : 802.11 EEPROM access */
#define HostCmd_CMD_802_11_EEPROM_ACCESS     0x0059
/** Host Command ID : 802.11 associate */
#define HostCmd_CMD_802_11_ASSOCIATE         0x0012
/** Host Command ID : 802.11 set WEP */
#define HostCmd_CMD_802_11_SET_WEP           0x0013
/** Host Command ID : 802.11 SNMP MIB */
#define HostCmd_CMD_802_11_SNMP_MIB          0x0016

/** Card Event definition : Dummy host wakeup signal */
#define EVENT_DUMMY_HOST_WAKEUP_SIGNAL  0x00000001
/** Card Event definition : Link lost with scan */
#define EVENT_LINK_LOST_WITH_SCAN       0x00000002
/** Card Event definition : Link lost */
#define EVENT_LINK_LOST                 0x00000003
/** Card Event definition : Link sensed */
#define EVENT_LINK_SENSED               0x00000004
/** Card Event definition : MIB changed */
#define EVENT_MIB_CHANGED               0x00000006
/** Card Event definition : Init done */
#define EVENT_INIT_DONE                 0x00000007
/** Card Event definition : Deauthenticated */
#define EVENT_DEAUTHENTICATED           0x00000008
/** Card Event definition : Disassociated */
#define EVENT_DISASSOCIATED             0x00000009
```

The whole process about command processing as follows. The driver handles the command from a user-space process such as ck5050, wpa_supplicant and initializes a structure HostCmd_DS_COMMAND by the function wlan_prepare_cmd(). The last argument pdata_buf points to a related structure that contains the necessary information to initialize the structure HostCmd_DS_COMMAND. The function wlan_process_cmdresp() is responsible for handling the command response from the chip and copying back the results to the structure references by pdata_buf.

```
int
wlan_prepare_cmd(wlan_private * priv,
                 u16 cmd_no,
                 u16 cmd_action,
                 u16 wait_option, WLAN_OID cmd_oid, void *pdata_buf);
```

The vulnerability exists in the function wlan_process_cmdresp() when the driver is processing the response of command HostCmd_CMD_GET_MEM. The function wlan_process_cmdresp() not check if the member size of structure HostCmd_DS_COMMAND is valid, which results in a buffer overflow when copying the data from structure HostCmd_DS_COMMAND to other place.

Code Execute in Wi-Fi Chip

Obviously, the vulnerability in firmware is a heap overflow. To utilize this vulnerability to gain code execution in the Wi-Fi chip, we need to figure out how the function memcpy() corrupted the memory, what could happen after triggering the vulnerability, and where the crash happens.

To trigger the vulnerability, the length of action header should be less than 4, and we must provide the correct dialog token in action frame, which means the length passed to memcpy() must be 0xffffffff. The source address is fixed because the source buffer allocates from memory pool pool_start_id_rmlmebuf, which has only one block. The destination buffer allocates from memory pool pool_start_id_tx. So the destination address could be one of the four addresses.

Source address	0xC0016478
Destination address	0xC000DC9B 0xC000E49B 0xC000EC9B 0xC000F49B
Length	0xFFFFFFFF

The source address and destination address locate in RAM region 0xC0000000~0xC003FFFF, but the address range from 0xC0000000 to 0xCFFFFFFF is valid. So, the results of reading or writing to these memory areas are the same.

0xC0000000~0xC0040000
0xC0040000~0xC0080000
0xC0080000~0xC00C0000
...

Because the memory region from 0xC0000000 to 0xCFFFFFFF is readable and writable, the process of copying is almost impossible to reach the boundary of the memory region. After 0x40000 bytes copied, the memory can be considered as shifted a distance once. In this process, some data could be overwritten and lost.

The CPU in 88w8688 contains only one core, so the chip may not crash during the execution of copying until an interrupt occurs. Since memory already corrupted by the vulnerability, in most cases, the chip crashed in the interrupt handlers.

The interrupt controller provides a simple firmware interface to the interrupt system. When an interrupt occurs, the firmware gets the interrupt event from the register of the interrupt controller and invokes the related interrupt handler.

Interrupt	Interrupt Source
Intr[18]	Wireless Encryption Unit (WEU) Interrupt
Intr[17]	Internal Bus Advance Encryption Unit (AEU) Interrupt
Intr[16]	DMA Control Unit (DCU) Channel 1 Interrupt
Intr[15]	DCU Channel 0 Interrupt
Intr[14]	AIU Interrupt
Intr[13]	Clocked Serial Unit (CSU) Interrupt
Intr[12]	GPIO Unit (GPU) Interrupt
Intr[11]	Serial Interface Unit (UART) Interrupt
Intr[10]	WLAN MAC Control Unit (MCU) interrupt
Intr[8]	Host Interface Unit (HIU) Interrupt
Intr[7:4]	General Purpose Timer Unit (RTU) Interrupt
Intr[3]	CommTx Interrupt
Intr[2]	CommRx Interrupt
Intr[1]	Firmware Enabled Programmed Interrupt
Intr[0]	FIQ Source Status Interrupt

There are many interrupt sources, so the chip can crash at many places after triggering the vulnerability.

One possibility is that the interrupt comes from 0x15, then the function 0x26580 be called. There is a link list pointer at 0xC000CC08. The value of this pointer could be overwritten after triggering the vulnerability. However, the manipulation of the link list may not be able to give us the chance to gain code execution.

```
void sub_26580()
{
  int p; // r4
  _DWORD *head; // r5
  int v3; // [sp+0h] [bp-18h]

  v3 = os_DisableInterupt();
  p = dword_C000CC08;
  head = dword_C000CC08;
  while ( p )
  {
    if ( ((*(p + 12))-- - 1) & 0xFFFF )
    {
      head = p;
    }
    else
    {
      if ( *(p + 14) )
      {
        head = p;
        *(p + 12) = *(p + 14);
      }
      else
      {
        if ( p == dword_C000CC08 )
        {
          head = *p;
          dword_C000CC08 = *p;
        }
        else
        {
          *head = *p;
        }
        *(p + 4) = 0;
      }
      if ( !*(p + 8) )
        msg_write_TimerCbMsg(p);
    }
    p = *p;
  }
  os_EnableInterupt(v3);
}
```

Another crash happens in the interrupt handler of the Timer Interrupt. The handler does thread switching sometimes, and another task could resume running, which means the process of copying can be suspended temporarily and the chip crash during other tasks running. In this situation, the firmware crashed in function 0x4D75C usually.

```
void __fastcall fun_4D75C(unsigned int a1)
{
  unsigned int v1; // r5@1
  char *v2; // r0@1
  int v3; // r1@4

  v1 = a1;
  os_semaphore_get(dword_C000D7DC, -1);
  v2 = &unk_C0025C7C + 16 * v1;
  if ( *(v2 + 2) || *(v2 + 3) )
  {
    os_semaphore_put(dword_C000D7DC);
  }
  else
  {
    v3 = dword_C0025CD8;
    *(dword_C0025CD8 + 8) = v2;
    dword_C0025CD8 = &unk_C0025C7C + 16 * v1;
    *(v2 + 3) = v3;
    *(v2 + 2) = &unk_C0025CCC;
    os_semaphore_put(dword_C000D7DC);
    sub_42DE0(v1);
  }
}
```

The function read a pointer at 0xC000D7DC, which points to structure TX_SEMAPHORE. After triggering the vulnerability, we can overwrite the pointer to our fake TX_SEMAPHORE structure.

```
typedef struct TX_SEMAPHORE_STRUCT
{
    ULONG       tx_semaphore_id;
    CHAR_PTR    tx_semaphore_name;
    ULONG       tx_semaphore_count;
    struct TX_THREAD_STRUCT  *tx_semaphore_suspension_list;
    ULONG                    tx_semaphore_suspended_count;
    struct TX_SEMAPHORE_STRUCT *tx_semaphore_created_next;
    struct TX_SEMAPHORE_STRUCT *tx_semaphore_created_previous;
} TX_SEMAPHORE;
```

If the member tx_semaphore_suspension_list also points to our fake TX_THREAD_STRUCT structure, when the function _tx_semaphore_put() update the link of the adjacent threads in TX_THREAD_STRUCT structure, we can get a chance to “write anything anywhere.”

```
thread_ptr =  semaphore_ptr -> tx_semaphore_suspension_list;
if (thread_ptr)
{
    /* Remove the suspended thread from the list.  */
    /* See if this is the only suspended thread on the list.  */
    if (thread_ptr == thread_ptr -> tx_suspended_next)
    {
        /* Yes, the only suspended thread.  */
        /* Update the head pointer.  */
        semaphore_ptr -> tx_semaphore_suspension_list =  TX_NULL;
    }
    else
    {
        /* At least one more thread is on the same expiration list.  */
        /* Update the list head pointer.  */
        semaphore_ptr -> tx_semaphore_suspension_list =  thread_ptr -> tx_suspended_next;
        /* Update the links of the adjacent threads.  */
        (thread_ptr -> tx_suspended_next) -> tx_suspended_previous =
                                        thread_ptr -> tx_suspended_previous;
        (thread_ptr -> tx_suspended_previous) -> tx_suspended_next =
                                        thread_ptr -> tx_suspended_next;
    }
```

We can directly overwrite the next instruction after “BL os_semaphore_put” with a jump instruction to archive code execute as the memory in ITCM is RWX. The difficulty lies in we need to spray both TX_SEMAPHORE structure and TX_THREAD_STRUCT structure in memory. We also need to make sure the pointer tx_semaphore_suspension_list in structure TX_SEMAPHORE points to our fake TX_THREAD_STRUCT structure. These conditions can be satisfied, but the success rate is very low.

We mainly focus on the third crash place, in the handler of MCU interrupts. The pointer g_interface_sdio points to structure struct_interface can be overwritten.

```
struct struct_interface
{
  int field_0;
  struct struct_interface *next;
  char *name_ptr;
  int sdio_idx;
  int fun_enable;
  int funE;
  int funF;
  int funD;
  int funA;
  int funB; // 0x24
  int funG;
  int field_2C;
};
```

The function pointer funB in this structure will be invoked in this function. If the pointer g_interface_sdio overwrited, arbitrary code execution can be achieved.

```
int __fastcall interface_call_funB(struct_interface *a1, int a2, int a3)
{
  int (__cdecl *funB)(int, int, int); // r3
  int ret; // r0

  if ( a1 && (funB = a1->funB) != 0 )
    ret = funB(a1, a2, a3);
  else
    ret = 1;
  return ret;
}
```

Here is the register dump when instruction “BX R3” executes in function interface_call_funB(). In this dump, g_interface_sdio overwrited by 0xabcd1211.

```
LOG_BP_M0_CPSR        : 0xa000009b
LOG_BP_M0_SP          : 0x5fec8
LOG_BP_M0_LR          : 0x3cd50
LOG_BP_M0_SPSP        : 0xa00000b2
LOG_BP_M1_CPSR        : 0xa0000092
LOG_BP_M1_SP          : 0x5536c
LOG_BP_M1_LR          : 0x4e3d5
LOG_BP_M1_SPSP        : 0xa0000013
LOG_BP_M2_CPSR        : 0
LOG_BP_M2_SP          : 0x58cb8
LOG_BP_M2_LR          : 0x40082e8
LOG_BP_M2_SPSP        : 0
LOG_BP_R1             : 0x1c
LOG_BP_R2             : 0
LOG_BP_R3             : 0xefdeadbe
LOG_BP_R4             : 0x40c0800
LOG_BP_R5             : 0
LOG_BP_R6             : 0x8000a500
LOG_BP_R7             : 0x8000a540
LOG_BP_R8             : 0x140
LOG_BP_R9             : 0x58cb0
LOG_BP_R10            : 0x40082e8
LOG_BP_FP             : 0
LOG_BP_IP             : 0x8c223fa3
LOG_BP_R0             : 0xabcd1211
```

The function interface_call_funB() called by the handler of MACMCU interrupt at 0x4E3D0.

```
v3 = interface_call_funB(g_interface_sdio, 28, 0) | v2 & 0x8880 | byte_4005361;// 0x4E3D0: BL interface_callB
dword_4000008 |= v3 | interface_call_funB(g_interface_sdio, 9, 0);
```

After the source address of copying reach the address 0xC0040000, the whole memory can be considered as shifted a distance once. After the source address of copying reach the address 0xC0080000, the whole memory shifted twice. The distance could be as follows.

```
0xC0016478-0xC000DC9B=0x87DD
0xC0016478-0xC000E49B=0x7FDD
0xC0016478-0xC000EC9B=0x77DD
0xC0016478-0xC000F49B=0x6FDD
```

After trigger the vulnerability, in most cases, the memory will be shifted 3~5 times when interrupt occurs. The pointer g_interface_sdio at address 0xC000B818, so g_interface_sdio can be overwritten by the data at these addresses.

```
0xC000B818+0x87DD*1=0xC0013FF5
0xC000B818+0x87DD*2=0xC001C7D2
0xC000B818+0x87DD*3=0xC0024FAF
0xC000B818+0x87DD*4=0xC002D78C
…
0xC000B818+0x7FDD*1=0xC00137F5
0xC000B818+0x7FDD*2=0xC001B7D2
0xC000B818+0x7FDD*3=0xC00237AF
0xC000B818+0x7FDD*4=0xC004B700
…
0xC000B818+0x77DD*1=0xC0012FF5
0xC000B818+0x77DD*2=0xC001A7D2
0xC000B818+0x77DD*3=0xC0021FAF
0xC000B818+0x77DD*4=0xC002978C
…
0xC000B818+0x6FDD*1=0xC00127F5
0xC000B818+0x6FDD*2=0xC00197D2
0xC000B818+0x6FDD*3=0xC00207AF
0xC000B818+0x6FDD*4=0xC002778C
…
```

The addresses 0xC0024FAF, 0xC00237AF and 0xC0021FAF located in a huge DMA buffer 0xC0021F90~0xC0025790 which is used for storing 802.11 Data Frame received by Wi-Fi chip temporarily. So, this huge buffer can be used to spray with fake pointers.

```
DMA_desc_struct <       0,         0,         0,         0,         0, \
                        ; DATA XREF: sub_233F0+16↑o
                        ; app:off_2360C↑o ...
                0x8000A434, 0x8000A448, DMA_fun_unknown+1>; 0
DMA_desc_struct <       1,         0,         0,         0,         0, \
                0x8000A438, 0x8000A44C, DMA_fun_unknown+1>; 1
DMA_desc_struct <       2,         0,         0,         0,         0, \
                0x8000A43C, 0x8000A450, DMA_fun_unknown+1>; 2
DMA_desc_struct <       3, 0xC0021F90, 0xC0025790,   0x3800,        0,\
                0x8000A430, 0x8000A444, DMA_fun_unknown+1>; 3
DMA_desc_struct <       4, 0xC0021390, 0xC0021F90,     0xC00,        0,\
                0x8000A440, 0x8000A454, DMA_fun_unknown+1>; 4
```

To spray our fake pointers in memory, we can send many normal 802.11 Data Frame full of fake pointers to Wi-Fi chip. The DMA buffer is so huge that we can directly spray our shellcode in it. To improve the success rate of exploiting, we used egg-hunters to search for our shellcode.

PTR	PTR	PTR	PTR	PTR	PTR	PTR	PTR	PTR	PTR	Egg Hunter

PTR	PTR	Shellcode

If we successfully overwrote g_interface_sdio, the shellcode or egg hunter can very close to 0xC000B818. The fake pointer we used is 0x41954 because there is a pointer 0xC000B991 at address 0x41954+0x24. Then, we can hijack \$PC to 0xC000B991. At the same time, the pointer 0x41954 can be recognized as normal instructions.

```
54 19 ADDS              R4, R2, R5
04 00 MOVS              R4, R0
```

We got about a 25% success rate to achieve code execution in this method.

Attack Host System

The vulnerability in kernel driver can be trigger by sending data from chip through SDIO interface.

The command HostCmd_CMD_GET_MEM initialize by function wlan_get_firmware_mem() in normal case.

```
if (!(buf = kmalloc(MRVDRV_SIZE_OF_CMD_BUFFER, GFP_KERNEL))) {
    PRINTM(INFO, "allocate  buffer failed!\n");
    LEAVE();
    return -ENOMEM;
}
memset(buf, 0, MRVDRV_SIZE_OF_CMD_BUFFER);

ret = wlan_prepare_cmd(priv,
                       HostCmd_CMD_GET_MEM, 0,
                       HostCmd_OPTION_WAITFORRSP, 0, buf);
if (!ret) {
    wrq->u.data.length = pFwData->size;
    if (copy_to_user
        (wrq->u.data.pointer, (u8 *) pGetMem, wrq->u.data.length)) {
        PRINTM(INFO, "Get Mem: copy to user failed\n");
        ret = -EFAULT;
        goto memexit;
    }
}
```

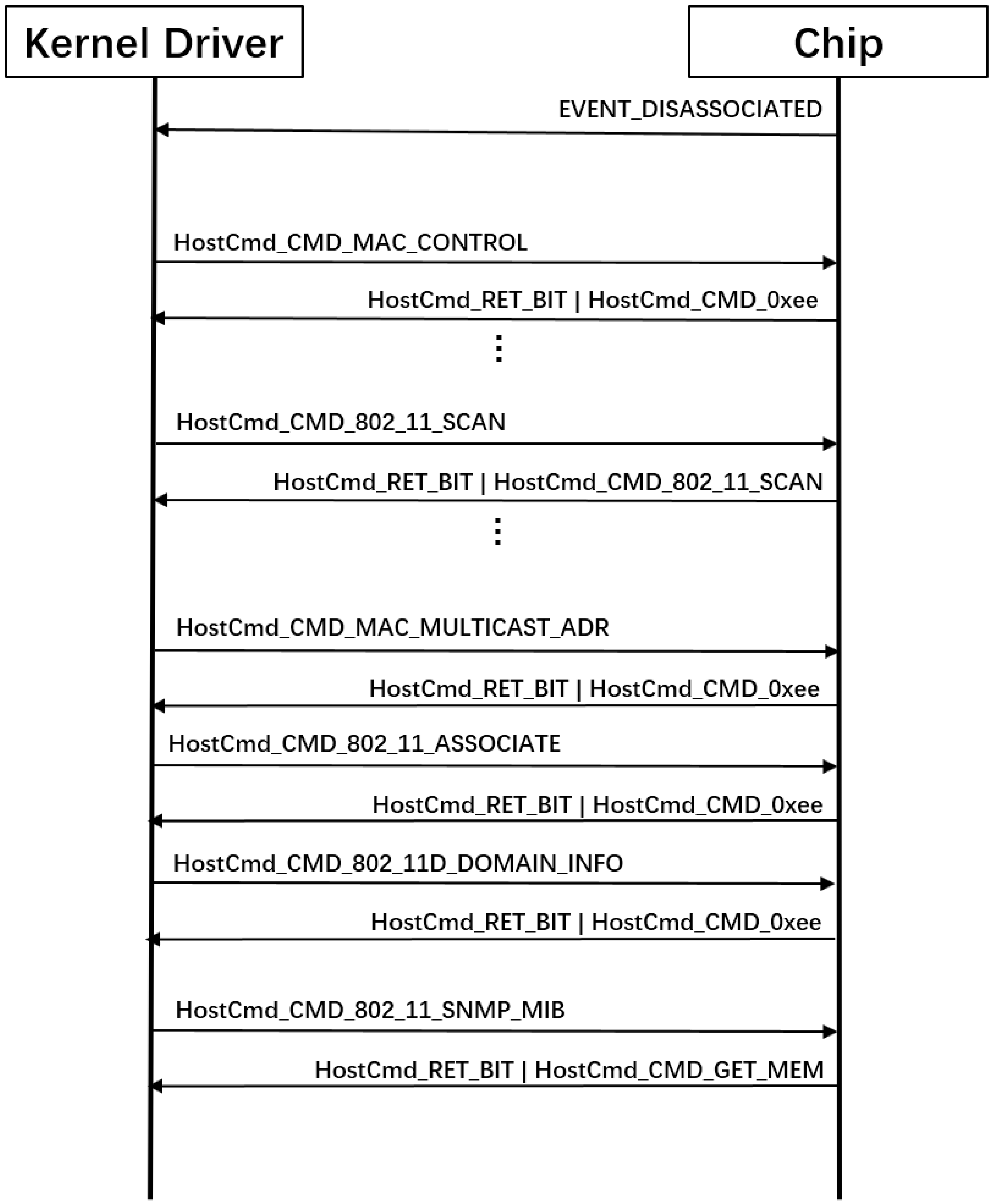
In this case, pdata_buf points to the buffer allocated by the function kmalloc(), which means it is a kernel heap overflow. The function wlan_get_firmware_mem() cannot be called in the real environment, and heap overflow is hard to exploit.

However, a compromised chip can return the result with a different command id after receiving a command. Therefore, the vulnerability can be triggered during the process of many command processing. In this situation, the vulnerability can be heap overflow or stack overflow depending on where pdata_buf points to. We found the function wlan_enable_11d(), which used the address of local variable enable as pdata_buf. Thus, we can trigger a stack buffer overflow.

```
int
wlan_enable_11d(wlan_private * priv, state_11d_t flag)
{
    wlan_adapter *Adapter = priv->adapter;
    wlan_802_11d_state_t *state = &Adapter->State11D;
    int ret;
    state_11d_t enable = flag;

    ENTER();

    state->Enable11D = flag;

    /* send cmd to FW to enable/disable 11D function in FW */
    ret = wlan_prepare_cmd(priv,
                           HostCmd_CMD_802_11_SNMP_MIB,
                           HostCmd_ACT_GEN_SET,
                           HostCmd_OPTION_WAITFORRSP, Dot11D_i, &enable);
    if (ret)
```

The function wlan_enable_11d() called by wlan_11h_process_join(). Obviously, HostCmd_CMD_802_11_SNMP_MIB used in the process of associating with AP. The vulnerability in firmware only can be trigger when Parrot already connects to an AP. When we get code execution in the chip, Parrot already joined an AP. To trigger the stack buffer overflow in wlan_enable_11d(), the compromised chip needs to deceive the kernel driver that the chip disconnects from AP. Then, a reconnection launched by the driver and the command HostCmd_CMD_802_11_SNMP_MIB sent to firmware in function wlan_enable_11d(). Therefore, to launch the reconnection, the chip only needs to send event EVENT_DISASSOCIATED to the driver.

After triggering the vulnerability and get code execution in chip, the chip cannot work properly anymore, so our shellcode running in chip need to handle a series of commands when Parrot is trying to reconnect to original AP. The only command we need to handle is HostCmd_CMD_802_11_SCAN before the command HostCmd_CMD_802_11_SNMP_MIB comes. Below is the whole process from disassociation to trigger kernel driver vulnerability.



The event and command packet can be sent directly by operating the register SDIO_CardStatus and SDIO_SQReadBaseAddress0. The register SDIO_SQWriteBaseAddress0 at 0x80000114 is useful for processing the data received from the kernel driver.

Command Execute in Linux System

As Linux Kernel 2.6.36 does not support NX, it’s possible to execute the shellcode on stack directly. In the meantime, the type of size in structure HostCmd_DS_COMMAND is u16, so the shellcode can be big enough to do lots of things.

After triggered vulnerability and controlled \$PC, \$R7 points to the kernel stack. It is very convenient to jump to the shellcode.

The function run_linux_cmd in shellcode called Usermode Helper API to execute Linux commands.

Get Shell Remotely

After triggering the vulnerability in chip, the whole RAM region corrupted, and the firmware cannot work anymore. Besides, the kernel stack is corrupted and needs to be repaired.

To make the wireless function of Parrot works again properly, we did these things:

1. After sending the kernel payload through the SDIO interface, we reset the chip by running the following code. Later, the kernel driver finds the chip and redownload the firmware.

```
*(unsigned int *)0x8000201c|=2;
*(unsigned int *)0x8000a514=0;
*(unsigned int *)0x80003034=1;
```

2. Call kernel function rtnl_unlock() in shellcode function fun_ret() to unlock rtnl_mutex which locked before wlan_enable_11d() called, or the wireless function in Linux will hangs, result in Parrot reboot by CID.

3. Call kernel function do_exit() in shellcode function fun_ret() to kill the user-mode process wpa_supplicant and restart it, so we don't need to repair the kernel stack.

4. Kill process ck5050 and start again, or ck5050 segment fault due to chip reset, result in Parrot reboot by CID.

To get shell remotely, we force Parrot to connect to our AP and alter iptables rules. Then, the shell listened on port 23 can be reached.

Finally, the success rate of getting a shell is about 10%.

Complete Exploit process

1. The attacker sends DEAUTH frames to all the AP nearby.
2. When Tesla reconnects to AP, the attacker gets the MAC address of Tesla.
3. Spray the fake pointer, then trigger the vulnerability in firmware by directly send corrupt Action Frame.
4. The function memcpy() executed until interrupt occurs.

5. Gain code execution in the Wi-Fi chip.

6. Stage 1 shellcode sends the event EVENT_DISASSOCIATED to the driver.

7. Stage 1 shellcode handles some commands and waits for the command HostCmd_CMD_802_11_SNMP_MIB.

8. Stage 1 shellcode sends the payload to trigger the kernel stack overflow through the SDIO interface.

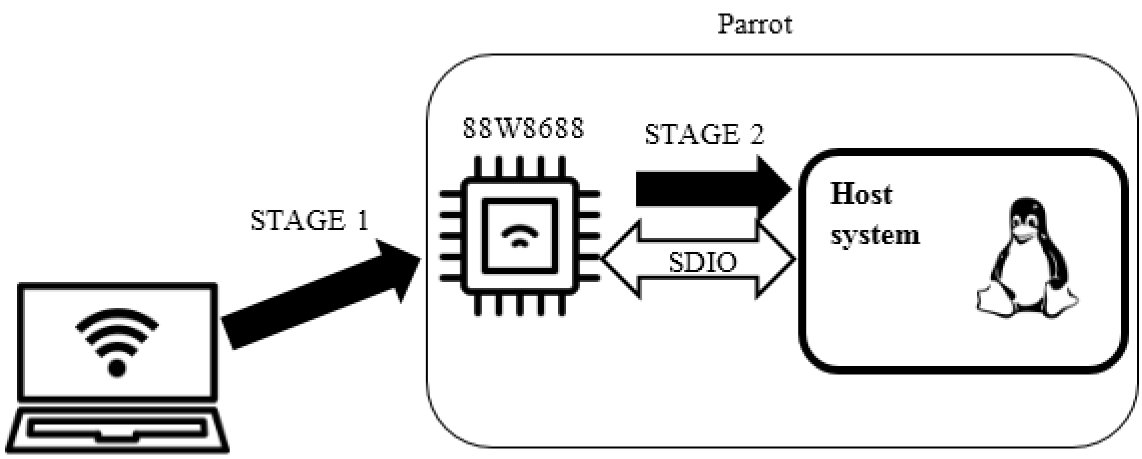
9. Stage 2 shellcode executed and invoke the kernel function call_usermodehelper().

10. Linux system command executed and try to fix the wireless function of Parrot.

11. Attacker setups an AP and a DHCP server in this AP

12. Linux system command forces the Parrot to join our AP and alter the iptables rules.

13. The attacker can telnet to port 23 on Parrot.



Demo Video

Conclusion

In this article, we presented the details of the vulnerability in the firmware and the vulnerability in the Marvell kernel driver and explained how to utilize these two vulnerabilities to compromise the Parrot Linux system by just sending malicious packets from a normal Wi-Fi dongle.

Responsible disclosure

All the two vulnerabilities we presented above are reported to Tesla in March 2019. Tesla already fixed them in version 2019.36.2, and the Marvell also has deployed a fix and published a security advisory[4] to the issue. The disclosure of the vulnerability research report had been communicated to Tesla, and Tesla is aware of our release.

You can track the issue from links below:

1. https://www.cnvd.org.cn/flaw/show/CNVD-2019-44105

2. http://www.cnnvd.org.cn/web/xxk/ldxqById.tag?CNNVD=CNNVD-201911-1040

3. http://www.cnnvd.org.cn/web/xxk/ldxqById.tag?CNNVD=CNNVD-201911-1038

4. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13581

5. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13582

References

[1] https://fccid.io/RKXFC6050W/Users-Manual/user-manual-1707044

[2] https://www.marvell.com/wireless/88w8688/

[3] https://www.marvell.com/wireless/assets/Marvell-88W8688-SoC.pdf

[4] https://www.marvell.com/documents/ioaj5dntk2ubykssa78s/

#CarHacking

OLDER

TenSec 2019

TAG CLOUD

CarHacking KeenLab TenSec Virtualization Windows

ARCHIVES

January 2020
May 2019
March 2019
July 2018
May 2018
April 2018
July 2017
January 2017
November 2016
September 2016
July 2016
June 2016
May 2016

RECENT POSTS

Exploiting Wi-Fi Stack on Tesla Model S
TenSec 2019
Tencent Keen Security Lab: Experimental Security Research of Tesla Autopilot

Exploiting iOS 11.0-11.3.1 Multi-path-TCP:A walk through
New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars