Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1)

Archived: 2026-04-05 20:38:13 UTC

Posted by Gal Beniamini, Project Zero

It's a well understood fact that platform security is an integral part of the security of complex systems. For mobile devices, this statement rings even truer; modern mobile platforms include multiple processing units, all elaborately communicating with one another. While the code running on the application processor (AP) has been the subject of much research, other components have seldom received the same scrutiny.

Over the years, as a result of the focused attention by security folk, the defenses of code running on the application processor have been reinforced. Taking Android as a case study, this includes hardening the operating system, improving the security of applications, and introducing incremental security enhancements affecting the entire system. All positive improvements, no doubt. However, attackers tend to follow the path of least resistance. Improving the security of one component will inevitably cause some attackers to start looking elsewhere for an easier point of entry.

In this two-part blog series, we'll explore the exposed attack surface introduced by Broadcom's Wi-Fi SoC on mobile devices. Specifically, we'll focus our attention on devices running Android, although a vast amount of this research applies to other systems including the same Wi-Fi SoCs. The first blog post will focus on exploring the Wi-Fi SoC itself; we'll discover and exploit vulnerabilities which will allow us to remotely gain code execution on the chip. In the second blog post, we'll further elevate our privileges from the SoC into the the operating system's kernel. Chaining the two together, we'll demonstrate full device takeover by Wi-Fi proximity alone, requiring no user interaction.

We'll focus on Broadcom's Wi-Fi SoCs since they are the most common Wi-Fi chipset used on mobile devices. A partial list of devices which make use of this platform includes the Nexus 5, 6 and 6P, most Samsung flagship devices, and all iPhones since the iPhone 4. For the purpose of this blog post, we'll demonstrate a Wi-Fi remote code execution exploit on a fully updated (at the time, now fixed) Nexus 6P, running Android 7.1.1 version NUF26K.

All the vulnerabilities in the post have been disclosed to Broadcom. Broadcom has been incredibly responsive and helpful, both in fixing the vulnerabilities and making the fixes available to affected vendors. For a complete timeline, see the bug tracker entries. They've also been very open to discussions relating to the security of the Wi-Fi SoC.

I would like to thank Thomas Dullien (@halvarflake) for helping boot up the research, for the productive brainstorming, and for helping search the literature for any relevant clues. I'd also like to thank my colleagues in the London office for helping make sense of the exploitation constraints, and for listening to my ramblings.

Why-Fi?

In the past decade, the use of Wi-Fi has become commonplace on mobile devices. Gradually, Wi-Fi has evolved into a formidable set of specifications—some detailing the physical layer, others focusing on the MAC layer. In order to deal with this increased complexity, vendors have started producing "FullMAC" Wi-Fi SoCs.

In essence, these are small SoCs that perform all the PHY, MAC and MAC SubLayer Management Entity (MLME) processing on their own, allowing the operating system to abstract itself away from the complex (and sometimes chip-specific) features related to Wi-Fi. The introduction of Wi-Fi FullMAC chips has also improved the power consumption of mobile devices, since much of the processing is done on a low-power SoC instead of the power-hungry application processor. Perhaps most importantly, FullMAC chips are much easier to integrate, as they implement the MLME within their firmware, reducing the complexity on the host's side.

All that said and done, the introduction of Wi-Fi FullMAC chips does not come without a cost. Introducing these new pieces of hardware, running proprietary and complex code bases, may weaken the overall security of the devices and introduce vulnerabilities which could compromise the entire system.

Exploring the Platform

To start off our research, we'll need to find some way to explore the Wi-Fi chip. Luckily, Cypress has recently acquired Broadcom's Wireless IOT business, and have published many of the datasheets related to Broadcom's Wi-Fi chipsets (albeit for a slightly older SoC, the BCM4339). Reading through the datasheet, we gain some insight into the hardware architecture behind the Wi-Fi chipset.

Specifically, we can see that there's an ARM Cortex R4 core, which runs all the logic for handling and processing frames. Moreover, the datasheet reveals that the ARM core has 640KB of ROM used to hold the firmware's code, and 768KB of RAM which is used for data processing (e.g., heap) and to store patches to firmware code.

To start analysing the code running on the ARM core, we'll need to extract the contents of the ROM, and to locate the data that is loaded into RAM.

Let's start by tackling the second problem first - where is the data that's loaded into the ARM core's RAM? Since this data is not present in ROM, it must be loaded externally when the chip first powers on. Therefore, by reading through the initialisation code in the host's driver, we should be able to locate the file containing the RAM's contents. Indeed, going over the driver's code, we find the BCMDHD_FW_PATH config, which is used to denote the location of the file whose contents are uploaded to RAM by the driver.

So what about the ROM's contents? One way to extract the ROM would be to use the host driver's chip memory access capabilities (via PIO over SDIO or PCIe) to read the ROM's contents directly. However, doing so would require modifying the driver to enable us to issue the commands needed to dump the ROM. Another way to retrieve the ROM would be to load our own modified firmware file into RAM, into which we'll insert a small stub that can be used to dump the ROM's memory range. Luckily, none of these approaches is actually needed in this case; Broadcom provides an extremely powerful command-line utility called dhdutil, which can be used to interact with the chip via the bcmdhd driver.

Among the various capabilities this utility supports, it also allows us to directly read and write memory on the dongle by issuing a special command - “membytes”. Since we already know the size of the ROM (from the datasheet), we can just use the membytes command to read the ROM’s contents directly. However, there’s one last question we need to answer first - where is the ROM located? According to the great research done by the folks behind NexMon, the ROM is loaded at address 0x0, and the RAM is loaded at address 0x180000 (while NexMon focused on BCM4339, this fact remains true for newer chips as well, such as the BCM4358).

Finally, putting all this together, we can acquire the RAM’s contents from the firmware file, dump the ROM using dhdutil, and combine the two into a single file which we can then start analysing in IDA.

Analysing the Firmware

Due to the relatively small size of the available memory (both ROM and RAM), Broadcom went to extreme efforts in order to conserve memory. For starters, they’ve stripped the symbols and most of the strings from the binary. This has the added bonus of making it slightly more cumbersome to reverse-engineer the firmware’s code. They’ve also opted for using the Thumb-2 instruction set exclusively, which allows for better code density. As a result, the ROM image on the BCM4358 is so tightly packed that it contains less than 300 unused bytes.

However, this is still not quite enough... Remember that the RAM has to accommodate the heap, stack and global data structures, as well as all the patches or modifications to ROM functions. Quite a tall order for a measly 768KB. To get around this, Broadcom has decided to place all the functions that are only used during the firmware’s initialisation in two special regions. Once the initialisation is completed, these regions are “reclaimed”, and are thereafter converted into heap chunks.

What’s more, heap chunks are interspersed between code and data structures in RAM - since the latter sometimes have alignment requirements (or are referenced directly from ROM, so they cannot be moved). The end result is that RAM is a jumbled mess of heap chunks, code and data structures.

After spending some time analysing the firmware, we can begin identifying at least a few strings containing function names and other hints, helping us get a grasp of the code base. Additionally, the NexMon researchers have released their gathered symbols corresponding to firmware on the BCM4339. We can apply the same symbols to the BCM4339’s firmware, and then use bindiff to correlate the symbol names in newer firmware versions for more recent chips.

Lastly, there’s one more trick in our hat - Broadcom produces SoftMAC chips in addition to the FullMAC SoCs we’re analysing. Since these SoftMAC chips don’t handle the MLME layer, their corresponding driver must perform that processing. As a result, much of Broadcom’s MLME processing code is included in the open-source SoftMAC driver - brcmsmac. While this won’t help us out with any of the chip-specific features or the more internal processing code, it does seem to share many utility functions with the firmware’s code.

Hunting for Bugs

Now that we have a grasp of the firmware's structure and have the means to analyse it, we can finally start hunting for bugs. But… Where should we start?

Even with all the tricks mentioned before, this is a relatively large and opaque binary, and strings or symbols are few and far between. One possibility would be to instrument the firmware in order to trace the code paths taken while a packet is received and processed. The Cortex R4 does, indeed, have debug registers which can be used to place breakpoints and inspect the code flow at various locations. Alternately, we could manually locate a set of functions which are used to parse and retrieve information from a received frame, and work our way backwards from there.

This is where familiarity with Wi-Fi comes in handy; Wi-Fi management frames encode most of their information in small "tagged" chunks of data, called Information Elements (IEs). These tagged chunks of data are structured as TLVs, where the tag and length fields are a single byte long.

Since a large portion of the information transferred in Wi-Fi frames (other than the data itself) is encoded using IEs, they make for good candidates from which we can work our way backwards. Moreover, as "tag" values are unique and standardised, we can use their values to help familiarise ourselves with the currently handled code flow.

Looking at the brcmsmac driver, we can see that there's a single function which Broadcom uses in order to extract IEs from a frame - bcm_parse_tlvs. After a brief search (by correlating hints from nearby strings), we find the same function in the firmware's ROM. Great.

Now we can start cross-referencing locations which call this function, and reverse each of these call-sites. While substantially easier than reversing every part of the firmware, this still takes a considerable amount of time (as the function has more than 110 cross-references, some to other wrapper functions which themselves are called from multiple locations).

After reverse engineering all of the call sites, I've found a few vulnerabilities related to the handling of information elements embedded in management frames.

Two of the vulnerabilities can be triggered when connecting to networks supporting wireless roaming features; 802.11r Fast BSS Transition (FT), or Cisco's CCKM roaming. On the one side, these vulnerabilities should be relatively straightforward to exploit - they are simple stack overflows. Moreover, the operating system running on the firmware (HNDRTE) does not use stack cookies, so there's no additional information leak or bypass required.

However, while these vulnerabilities may be comfortable to exploit, they require some set-up to get working. First, we'd need to broadcast Wi-Fi networks that support these features. 802.11r FT is an open(-ish) standard, and is implemented by hostapd. In contrast, CCKM is a proprietary standard (although some information can be found online). Figuring out how to emulate a CCKM network (or buying a CCKM-capable WLC from Cisco) would be cumbersome (or costly).

Additionally, we'd need to figure out which devices actually support the aforementioned features. Broadcom provides many features which can be licensed by customers -- not all features are present on all devices (in fact,

their corresponding patches probably wouldn't even fit in RAM).

Luckily, Broadcom makes it easy to distinguish which features are actually present in each firmware image. The last few bytes in the RAM contents downloaded to the chip contain the firmware's "version string". This string contains the date at which the firmware was compiled, the chip's revision, the firmware's version and a list of dash-delimited "tags". Each tag represents a feature that is supported by the firmware image. For example, here's the version string from the Nexus 6P:

4358a3-roml/pcie-ag-p2p-pno-aoe-pktfilter-keepalive-sr-mchan-pktctx-hostpp-lpc-pwropt-txbf-wl11u-mfp-betdls-amsdutx5g-txpwr-rcc-wepso-sarctrl-btcdyn-xorcsum-proxd-gscan-linkstat-ndoe-hs20sta-oobrev-hchk-logtrace-rmon-apf-d11status Version: 7.112.201.1 (r659325) CRC: 8c7aa795 Date: Tue 2016-09-13 15:05:58 PDT Ucode Ver: 963.317 FWID: 01-ba83502b

The presence of the 802.11r FT feature is indicated by the "fbt" tag. Similarly, support for CCKM is indicated by the "ccx" tag. Unfortunately, it seems that the Nexus 6P supports neither of these features. In fact, running a quick search for the "ccx" feature (CCKM support) on my own repository of Android firmware images revealed that this feature is not supported on any Nexus device, but is supported on a wide variety of Samsung flagship devices, a very partial list of which includes the Galaxy S7 (G930F, G930V), the Galaxy S7 Edge (G935F, G9350), the Galaxy S6 Edge (G925V) and many more.

So what about the other two vulnerabilities? Both of them relate to the implementation of Tunneled Direct Link Setup (TDLS). TDLS connections allow peers on a Wi-Fi network to exchange data between one another without passing it through the Access Point (AP), thus preventing congestion at the AP.

Support for TDLS in the firmware is indicated by the "betdls" and "tdls" tags. Searching through my firmware repository I can see that the vast majority of devices do, indeed, support TDLS. This includes all recent Nexus devices (Nexus 5, 6, 6P) and most Samsung flagships.

What's more, TDLS is specified as part of the 802.11z standard (requires IEEE subscription). Since all the information regarding TDLS is available, we could read the standard in order to gain familiarity with the relevant code paths in Broadcom's implementation. As an open standard, it is also supported by open-source supplicants, such as wpa_supplicant. As a result, we can inspect the implementation of the TDLS features in wpa_supplicant in order to further improve our understanding of the relevant code in the firmware.

Lastly, as we'll see later on, triggering these two vulnerabilities can be done by any peer on the Wi-Fi network, without requiring any action on the part of the device being attacked (and with no indication that such an attack is taking place). This makes these vulnerabilities all the more interesting to explore.

In any case, it seems like we've made our mind up! We're going to exploit the TDLS vulnerabilities. Before we do so, however, let's take a second to learn a little bit about TDLS, and the vulnerabilities discovered (skip this part it you're already familiar with TDLS).

802.11z TDLS 101

There are many use cases where two peers on the same Wi-Fi network wish to transfer large swaths of data between one another. For example, casting a video from your mobile device to your Chromecast would require

large amounts of data to be transmitted. In most cases, the Chromecast would be relatively nearby to the caster (after all, you'd probably be watching the screen to which you're casting). Therefore, it would seem wasteful to pass the entire data stream from the device to the AP, only to then pass it on to the Chromecast.

It's not just the increased latency of adding an additional hop (the AP) that will degrade the connection's quality. Passing such large amounts of data to the AP would also put a strain on the AP itself, cause congestion, and would degrade the Wi-Fi connectivity for all peers on the network.

This is where TDLS comes into play. TDLS is meant to provide a means of peer-to-peer communication on a Wi-Fi network that is AP-independant.

Over The Air

Let's start by familiarising ourselves with the structure of TDLS frames. As you may know, 802.11 frames use the "flags" field in order to indicate the "direction" in which a frame is travelling (from the client to the AP, AP to client, etc.). TDLS traffic co-opts the use of the flag values indicating traffic in an Ad-Hoc (IBSS) network (To-DS=0, From-DS=0).

Next, TDLS frames are identified by a special ethertype value - 0x890D. TDLS frames transmitted over Wi-Fi use a constant value in the "payload type" field, indicating that the payload has the following structure:

The category for TDLS frames is also set to a constant value. This leaves us with only one field which distinguishes between different TDLS frame types - the "action code". This 1-byte field indicates the kind of TDLS frame we're transmitting. This, in turn, controls the way in which the "payload" in interpreted by the receiving end.

High-Level Flow

Before two peers can establish a connection, they must first know about the existence of one another. This is called the "discovery" phase. A Wi-Fi client that wishes to discover TDLS-capable peers on the network, can do so by sending a "TDLS Discovery Request" frame to a peer. A TDLS-capable peer that receives this frame, responds by sending a "TDLS Discovery Response" frame. The request and response are correlated to one another using a 1-byte "dialog token".

Next, the peers may wish to set up a connection. To do so, they must perform a 3-way handshake. This handshake serves a dual purpose; first, it indicates that a connection is successfully established between the two peers. Second, it's used to derive the TDLS Peer Key (TPK), which secures the TDLS traffic between the peers.

Finally, once the connection is created, the two peers can exchange peer traffic between one another. When one of the peers wishes to tear-down the connection, they may do so by sending a “TDLS Teardown” frame. Upon reception of such a frame, the TDLS-peer will remove the connection and free up all the related resources.

Now that we know enough about TDLS, let’s take a closer look at the vulnerabilities at hand!

The Primitives

In order to ensure the integrity of messages transferred during the setup and teardown phases, the corresponding TDLS frames include Message Integrity Codes (MIC). For the setup phase, once the second handshake message (M2) is received, the TPK can be derived by both parties. Using the TPK, the TDLS-initiator can calculate a MIC over the contents of the third handshake frame, which can then be verified by the TDLS-responder.

So how can we find these calculations in the firmware’s code? Well, as luck would have it, some strings referring to TDLS were left-over in the firmware’s ROM, allowing us to quickly home in on the relevant functions.

After reverse-engineering much of the flow leading up to the processing of handling TDLS action frames, we finally reach the function responsible for handling TDLS Setup Confirm (PMK M3) frames. The function first performs some validations to ensure that the request is legitimate. It queries the internal data structures to ensure that a TDLS connection is indeed being set up with the requesting peer. Then, it verifies the Link-ID IE (by checking that its encoded BSSID matches that of the current network), and also verifies the 32-byte initiator nonce (“Snonce”) value (by comparing it to the stored initial nonce).

Once a certain degree of confidence is established that the request may indeed be legitimate, the function moves on to call an internal helper function, tasked with calculating the MIC and ensuring that it matches the one encoded in the frame. Quite helpfully, the firmware also includes the name for this function (“wlc_tdls_cal_mic_chk”).

After reverse-engineering the function, we arrive at the following approximate high-level logic:

```
1.  uint8_t* buffer = malloc(256);
2.  uint8_t* pos = buffer;
3.
4.  //Copying the initial (static) information
5.  uint8_t* linkid_ie = bcm_parse_tlvs(..., 101);
6.  memcpy(pos, linkid_ie + 0x8, 0x6);  pos += 0x6;            //Initiator MAC
7.  memcpy(pos, linkid_ie + 0xE, 0x6);  pos += 0x6;            //Responder MAC
8.  *pos = transaction_seq;            pos++;                 //TransactionSeq
9.  memcpy(pos, linkid_ie, 0x14);      pos += 0x14;           //LinkID-IE
10.
11. //Copying the RSN IE
```

```
12. uint8_t* rsn_ie = bcm_parse_tlvs(..., 48);
13. if (rsn_ie[1] + 2 + (pos - buffer) > 0xFF) {
14.     ... //Handle overflow
15. }
16. memcpy(pos, rsn_ie, rsn_ie[1] + 2); pos += rsn_ie[1] + 2;    //RSN-IE
17.
18. //Copying the remaining IEs
19. uint8_t* timeout_ie = bcm_parse_tlvs(..., 56);
20. uint8_t* ft_ie      = bcm_parse_tlvs(..., 55);
21. memcpy(pos, timeout_ie, 0x7);       pos += 0x7;             //Timeout Interval IE
22. memcpy(pos, ft_ie, 0x54);           pos += 0x54;            //Fast-Transition IE
```

As can be seen above, although the function verifies that the RSN IE's length does not exceed the allocated buffer's length (line 13), it fails to verify that the subsequent IEs also do not overflow the buffer. As such, setting the RSN IE's length to a large value (e.g., such that rsn_ie[1] + 2 + (pos - buffer) == 0xFF) will cause the Timeout Interval and Fast Transition IEs to be copied out-of-bounds, overflowing the buffer.

For example, assuming we set the length of the RSN IE (x) to its maximal possible value, 224, we arrive at the following placements of elements:

In this diagram, orange fields are those which are "irrelevant" for the overflow, since they are positioned within the buffer's bounds. Red fields indicate values that cannot be fully controlled by us, and green fields indicate values which are fully controllable.

For example, the Timeout Interval IE is verified prior to the MIC's calculation and only has a constrained set of allowed values, making it uncontrollable. Similarly, the FTIE's tag and length fields are constant, and therefore not controllable. Lastly, the 32-byte "Anonce" value is randomly chosen by the TDLS responder, placing it firmly out of our field of influence.

But the situation isn't that grim. In fact, several of the fields in the FTIE itself can be arbitrarily chosen - for example, the "Snonce" value is chosen by the TLDS-initiator during the first message in the handshake. Moreover, the "MIC Control" field in the FTIE can be freely chosen, since it is not verified prior to the execution of this function.

In any case, now that we've audited the MIC verification for the setup stage, let's turn our sights towards the MIC verification during the teardown stage. Perhaps the code is similarly broken there? Taking a look at the MIC calculation in the teardown stage ("wlc_tdls_cal_mic_chk"), we arrive at the following high-level logic:

```
1. uint8_t* buffer = malloc(256);
2. ...
3. uint8_t* linkid_ie = bcm_parse_tlvs(..., 101); //Link ID
4. memcpy(buffer, linkid_ie, 0x14);
```

```
5.  ...
6.  uint8_t* ft_ie = bcm_parse_tlvs(..., 55);
7.  memcpy(buffer + 0x18, ft_ie, ft_ie[1] + 2);    //Fast-Transition IE
```

Ah-ha, so once again a straightforward overflow; the FT-IE's length field is not verified to ensure that it doesn't exceed the length of the allocated buffer. This means that simply by providing a crafted FT-IE, we can trigger the overflow. Nevertheless, once again there are several verifications prior to triggering the vulnerable code path which limit our control on the overflowing elements. Let's try and plot the placement of elements during the overflow:

This seems much simpler - we don't need to worry ourselves about the values stored in the FTIE that are verified prior to the overflow, since they're all placed neatly within the buffer's range. Instead, the attacker controlled portion is simply spare data that is not subject to any verification, and can therefore be freely chosen by us. That said, the overflow's extent is quite limited - we can only overwrite at most 25 bytes beyond the range of the buffer.

Writing an Exploit

Investigating the Heap State

At long last we have a grasp of the primitives at hand. Now, it's time to test out whether our hypotheses match reality. To do so, we'll need a testbed that'll enable us to send crafted frames, triggering the overflows. Recall that wpa_supplicant is an open-source portable supplicant that fully supports TDLS. This makes it a prime candidate for our research platform. We could use wpa_supplicant as a base on top of which we'll craft our frames. That would save us the need to re-implement all the logic entailed in setting up and maintaining a TDLS connection.

To test out the vulnerabilities, we'll modify wpa_supplicant to allow us to send TDLS Teardown frames containing an overly-large FTIE. Going over wpa_supplicant's code, we can quickly identify the function in charge of generating and sending the teardown frame - wpa_tdls_send_teardown. By adding a few small changes to this function (in green) we should be able to trigger the overflow upon reception the teardown frame, causing 25 bytes of 0xAB to be written OOB:

```
static int wpa_tdls_send_teardown(struct wpa_sm *sm, const u8 *addr, u16 reason_code)

{

...

ftie = (struct wpa_tdls_ftie *) pos;

ftie->ie_type = WLAN_EID_FAST_BSS_TRANSITION;

ftie->ie_len = 255;

os_memset(pos + 2, 0x00, ftie->ie_len);
```

```
os_memset(pos + ftie->ie_len + 2 - 0x19, 0xAB, 0x19); //Overflowing with 0xAB

os_memcpy(ftie->Anonce, peer->rnonce, WPA_NONCE_LEN);

os_memcpy(ftie->Snonce, peer->inonce, WPA_NONCE_LEN);

pos += ftie->ie_len + 2;

...

}
```

Now we just need to interact with wpa_supplicant in order to setup and teardown a TDLS connection to our target device. Conveniently, wpa_supplicant supports many command interfaces, including a command-line utility called wpa_cli. This command line interface also supports several commands exposing TDLS functionality:

- TDLS_DISCOVER - Sends a “TDLS Discovery Request” frame and lists the response
- TDLS_SETUP - Creates a TDLS connection to the peer with the given MAC address
- TDLS_TEARDOWN - Tears down the TDLS connection to the peer with the given MAC

Indeed, after compiling wpa_supplicant with TDLS support (CONFIG_TDLS), setting up a network, and connecting our target device and our research platform to the network, we can see that issuing the TDLS_DISCOVER command works - we can indeed identify our peer.

Moving on, we can now send a TDLS_SETUP command, followed by our crafted TDLS_TEARDOWN. If everything adds up correctly, this should trigger the overflow. However, this raises a slightly more subtle question - how will we know when the overflow occurs? It may just so happen that the data we’re overflowing is unused. Alternately, it may be the case that when the firmware crashes, it just silently starts up again, leaving us none the wiser.

To answer this fully, we’ll need to understand the logic behind Broadcom’s heap implementation. Digging into the allocator’s logic, we find that it is extremely straightforward; it is a simple “best-fit” allocator, which performs forward and backward coalescing, and keeps a singly linked list of free chunks. When chunks are allocated, they are carved from the end (highest address) of the best-fitting free chunk (smallest chunk that is large enough). Heap chunks have the following structure:

(recall that the Cortex R4 is a 32-bit ARM processor, so all fields are stored in little-endian)

By reverse-engineering the allocator’s implementation, we can also find the location of the pointer to the head of the first free-chunk in RAM. Combining these two facts together, we can create a utility which, given a dump of the firmware’s RAM, can plot the current state of the heap’s freelist. Acquiring a snapshot of the firmware’s RAM can be easily achieved by using dhdutil’s “upload” command.

After writing a small visualiser script which walks over the heap's freelist and exports the its contents into dot, we can plot the state of the freelist using graphviz, like so:

Now, we can send out crafted TDLS_TEARDOWN frame, immediately take a snapshot of the firmware's RAM, and check the freelist for any signs of corruption:

Ah-ha! Indeed one of the chunks in the freelist suddenly has an exceptionally large size after tearing down the connection. Recall that since the allocator uses "best-fit", this means that subsequent allocations won't be placed in this block as long as other large enough free chunks exist. This also means that the firmware did not crash, and in fact continued to function correctly. Had we not visualised the state of the heap, we wouldn't have been able to determine anything had happened at all.

In any case, now that we've confirmed that the overflow does in fact occur, it's time to move to the next stage of exploitation. We need less crude tools in order to allow us to monitor the state of the heap during the setup and teardown processes. To this end, it would be advantageous to hook the malloc and free functions in the firmware, and to trace their arguments and return values.

First, we'll need to write a "patcher", which will allow us to insert hooks on given RAM-resident functions. It's important to note that both the malloc and free functions are both present in RAM (they are among the first functions in the RAM's code chunk). This allows us to freely re-write their prologues in order to introduce a branch to our own code. I've written a patcher which performs insertion of such hooks, allowing execution of small assembly stubs before and after the invocation of the hooked function.

In short, the patcher is fairly standard - it writes the patch's code to an unused region in RAM (the head of the largest free chunk in the heap), and then inserts a Thumb-2 wide branch (which is, coincidentally, perhaps the ugliest encoding for an opcode I've ever seen - see 4.6.12 T4) from the prologue of the hooked function to the hook itself.

Using our new patcher, we can now instrument the malloc and free functions in order to add traces allowing us to follow every operation occurring on the heap. These traces can then be read from the firmware's console buffer, by issuing dhdutil's "consoledump" command. Note that on some newer chips, like the BCM4358 on the Nexus 6P, this command fails. This is because Broadcom forgot to add the offset to the magic pointer in the firmware pointing to the console's data structure. You can fix this either by adding the correct offset to the driver (see debug_info_ptrs), or by writing the magic value and pointer to one of the probed memory addresses in the list.

In any case, you can find both the malloc and free hooks, and the associated scripts needed to parse the traces from the firmware, here.

Using the newly acquired traces, we can write a better visualiser, allowing us to trace the state of the heap throughout the setup and teardown phases. This visualiser will have visibility into every operation occurring on

the heap, offering far more granular data. I've written such a visualiser, which you can find here.

Without further ado, let's take a look at heap activity while establishing a TDLS connection:

The vertical axis denotes time - each line is a new heap state after a malloc or free operation. The horizontal axis denotes space - lower addresses are on the left, while higher addresses are on the right. Red blocks indicate chunks that are in-use, grey blocks indicate free chunks.

As we can clearly see above, establishing a TDLS connection is a messy operation. There are many allocations and deallocations, for regions both large and small. This abundance of noise doesn't bode well for us. Recall that the overflow during the setup stage is highly constrained, both in terms of the data being written, and in terms of the extent of the overflowing data. Moreover, the overflow occurs during one of the many allocations in the setup phase. This doesn't allow us much control over the state of the heap prior to triggering the overflow.

Taking a step back, however, we can observe a fairly surprising fact. Apart from the heap activity during the TDLS connection establishment, it seems like there is little to no activity on the heap whatsoever. In fact, it turns out that transmitted and received frames are drawn from a shared pool, instead of the heap. Not only that, but their processing doesn't incur a single heap operation - everything is done "in-place". Even when trying to intentionally cause allocations by sending random frames containing exotic bit combinations, the firmware's heap remains largely unaffected.

This is both a blessing and a curse. On the one hand, it means that the heap's structure is highly consistent. In the seldom events that data structures are allocated, they are immediately freed thereafter, restoring the heap to its original state. On the other hand, it means that our degree of control over the heap's structure is fairly limited. For the most part, whatever structure the heap has after the firmware's initialisation, is what we're going to have to work with (unless, of course, we find some primitive that will allow us to better shape the heap).

Perhaps we should take a look at the teardown stage instead? Indeed, activating the traces during the TDLS teardown stage reveals that there are very few allocations prior to triggering the overflow, so it seems like a much more convenient environment to explore.

While these in-depth traces are useful for getting a high-level view of the heap's state, they are rather difficult to decipher. In fact, in most cases it's sufficient to take a single snapshot of the heap and just visualise it, as we did earlier with the graphviz visualiser. In that case, let's improve our previous heap visualiser by allowing it to produce detailed graphical output, based on a single snapshot of the heap.

As we've seen earlier, we can "walk" over the freelist to extract the location and size of each free chunk. Moreover, we can deduce the location of in-use chunks by walking over the gaps between free chunks and reading the "size" field from each in-use chunk. I've written yet another visualiser that does just that - it simply produces a visualisation of the heap's state from a series of "snapshot" images.

Using this visualiser, we can now take a look at the state of the heap after setting up a TDLS connection. This will be the state of the heap we need to work with when we trigger the overflow during the teardown stage.

(Upper Layer: initial heap state, Bottom Layer: heap state after creating a TDLS connection) 

We can see that after setting up the TDLS connection, most of the heap's used chunks are consecutive, but also two holes are formed; one of size 0x11C, and another of size 0x124. Activating the traces for the teardown stage, we can see that the following allocations occur:

(29) malloc - size: 284, caller: 1828bb, res: 1f0404

(30) free - ptr: 1f0404

(31) malloc - size: 20, caller: 18c811, res: 1f1654

(32) malloc - size: 160, caller: 18c811, res: 1f0480

(33) malloc - size: 8, caller: 80eb, res: 1f2a44

(34) free - ptr: 1f2a44

(35) free - ptr: 1f1654

(36) free - ptr: 1f0480

(37) malloc - size: 256, caller: 7aa15, res: 1f0420

(38) malloc - size: 16, caller: 7aa23, res: 1f1658

The highlighted line denotes the allocation of the 256-byte buffer for the teardown frame's MIC calculation, that same one we can overflow using our vulnerability. Moreover, it seems as though the heap activity is quite low prior to sending the overflow frame. Combining the heap snapshot above with the trace file, we can deduce that the best-fitting chunk for the 256-byte buffer is in the 0x11C-byte hole. This means that using our 25-byte overflow we'll be able to overwrite:

1. The header of the next in-use chunk
2. A few bytes from the contents of the next in-use chunk

Let's take a closer look at the next in-use chunk and see whether there's any interesting information that we'd like to overwrite there:

Ah, so the next chunk is mostly empty, save for a couple of pointers near its head. Are these pointers of any use to us? Perhaps they are written to? Or freed at a later stage? We can find out by manually corrupting these pointers (pointing them at invalid memory addresses, such as 0xCDCDCDCD), and instrumenting the firmware's exception vector to see whether it crashes. Unfortunately, after many such attempts, it seems as though none of these pointers are in fact used.

This leaves us, therefore, with a single possibility - corrupting the “size” field of the in-use chunk. Recall that once the TDLS connection is torn down, the data structures relating to it are freed. Freeing an in-use chunk whose size we’ve corrupted could have many interesting consequences. For starters, if we reduce the size of the chunk, we can intentionally “leak” the tail end of the buffer, causing it to remain forever un-allocatable. Much more interestingly, however, we could set the chunk’s size to a larger value, thereby causing the next free operation to create a free chunk whose tail end overlaps another heap chunk.

Once a free chunk overlaps another heap chunk, subsequent allocations for which the overlapping free chunk is the best-fit will be carved from the end of the free chunk, thereby corrupting whatever fields reside at its tail. Before we start scheming, however, we need to confirm that we can create such a state (i.e., an overlapping chunk), after the teardown operation completes.

Creating an Overlapping Chunk

Recall that the MIC check is just one of many operations that take place when a TDLS connection is torn down. It may just so happen that by overwriting the next chunk’s size, once it is freed during the collection of the TDLS session’s data structures, it may become the best-fit for subsequent allocations during the teardown process. These allocations may then cause additional unintended corruptions, which will either leave the heap in a non-consistent state or even crash the firmware.

However, the search space for possible sizes isn’t that large - assuming we’re only interested in chunk sizes that are not larger than the RAM itself (for obvious reasons), we can simply enumerate each of the heap states produced by overwriting the “size” field of the next chunk with a given value and tearing down the connection. This can be automated by using a script on the sending (to perform the enumeration), while concurrently acquiring “snapshots” of RAM on the device, and observing their state (whether or not they are consistent, and whether the firmware managed to resume operation after the teardown).

Specifically, it would be highly advantageous if we were able to create a heap state whereby two free chunks overlap one another. In such a condition, allocations taken from one chunk, can be used to corrupt the “next” pointer of the other free chunk. This could be used, perhaps, to control the location of subsequent allocations - an interesting primitive in it’s own right.

In any case, after running through a few chunk sizes, tearing down the TDLS connection and observing the heap state, we come across quite an interesting resulting state! By overwriting the “size” field with the value 72 and tearing down the connection, we achieve the following heap state:

Great! So after tearing down the connection, we are left with a zero-sized free chunk, overlapping a different (larger) free chunk! This means that once an allocation is carved from the large chunk, it will corrupt the “size” and “next” fields of the smaller chunk. This could prove very useful - we could try and point the next free chunk at a memory address whose contents we’d like to modify. As long as the data in that address conforms with the

format of a free chunk, we might be able to persuade the heap to overwrite the memory at that address with subsequent allocations.

Finding a Controlled Allocation

To start exploring these possibilities, we'll first need to create a controlled allocation primitive, meaning we either control the size of the allocation, or it's contents, or (ideally) both. Recall that, as we've seen previously, it is in fact very hard to trigger allocations during the normal processing of the firmware - nearly all the processing is done in-place. Moreover, even for cases where data is allocated, its lifespan is very short; memory is immediately reclaimed once it's no longer used.

Be that as it may, we've already seen at least one set of data structures whose lifetime is controllable, and which contains multiple different pieces of information - the TDLS connection itself. The firmware must keep all the information pertaining to the TDLS connection as long as its active. Perhaps we could find some data structure relating to TDLS which could act as a good candidate for a controlled allocation?

To search for one, let's start by looking at the function handling each of the TDLS action frames - wlc_tdls_rcv_action_frame. The function starts by reading out the TDLS category and action code. Then, it routes the frame to the appropriate handler function, according to the received action code:

We can see that apart from the regular, specification-defined action codes, the firmware also supports an out-of-spec frame with an action code of 127. Anything out-of-spec is automatically suspect, so that might be as good a place as any to look for our primitive.

Indeed, digging into this function, we find out that it performs a rather curious task. First, it verifies that the first 3 bytes in the frame's contents match the Wi-Fi alliance OUI (50:6F:9A). Then, it retrieves the fourth byte of the frame, and uses it as a "command code". Currently, only two vendor-specific commands are implemented, commands #4 and #5. On a high-level; command #4 is used to send a tunneled probe request over the TDLS connection, and command #5 is used to send an "event" notification to the host, signalling that a "special" frame has arrived.

However, much more interestingly, we see that the implementation for command #4 seems relevant to our current pursuit. First, it does not require the existence of a TDLS connection in order to be processed! This allows us to send the frame even after tearing down the connection. Second, by activating heap traces during this function's execution and reverse-engineering its logic, we find that the function triggers the following high-level sequence of events:

1. if (A) { free(A); }

2. A = malloc(received_frame_size);

3. memcpy(A, received_frame, received_frame_size);

4. B = malloc(788);

5. free(B)

6. C = malloc(284);

7. free(C);

Great! So we get an allocation (A) with a controlled lifetime, a controlled size and controlled contents! What more could we possibly ask for?

There is one tiny snag, however. Modifying wpa_supplicant to send this crafted TDLS frame results in a resounding failure. While wpa_supplicant allows us to fully control many of the fields in the TDLS frames, it is only a supplicant, not an MLME implementation. This means that the corresponding MLME layer is responsible for composing and sending the actual TDLS frames.

On the setup I'm using for the attack platform, I have a laptop running Ubuntu 16.04, and a TP-Link TL-WN722N dongle. The dongle is a SoftMAC configuration, so the MLME layer in play is the one present in the Linux kernel, namely, the "cfg80211" configuration layer.

When wpa_supplicant wishes to create and send TDLS frames, it does so by sending special requests over Netlink, which are then handled by the cfg80211 framework, and subsequently passed to the SoftMAC layer, "mac80211". Regrettably, however, mac80211 is unable to process the special vendor frames, and simply rejects them. Nonetheless, this is just a minor inconvenience - I've written a few patches to mac80211 which add support for these special vendor frames. After applying these patches, re-compiling and booting the kernel, we are now able to send our crafted frames.

To allow for easier control over the vendor frames, I've also added support for a new command within wpa_supplicant's CLI - "TDLS_VNDR". This command allows us to send a crafted TDLS vendor frame with arbitrary data to any MAC address (regardless of whether a TDLS connection is established to that peer).

Putting It All Together

After creating two overlapping chunks, we can now use our controlled allocation primitive to allocate memory from the tail of the larger chunk, thereby pointing the smaller free chunk at a location of our choosing. Whichever location we choose, however, must have valid values for both the "size" and "next" fields, otherwise later calls to malloc and free may fail, possibly crashing the firmware. As a matter of fact, we've already seen perfect candidates to stand-in for free chunks - in-use chunks!

Recall that in-use chunks specify their size field at the same location free chunks do theirs. As for the "next" pointer, it is unused in free chunks, but is set to zero during the allocation of the chunk. This means that by corrupting the free list to point at an in-use chunk, we can trick the heap into thinking it's just another free chunk, which is coincidentally also the last chunk in the freelist. That's comfortable.

Now all we need to do is find an in-use chunk containing information that we'd like to overwrite. If we make that chunk the best-fitting chunk in the free list for a subsequent controlled allocation, we'll get our own data to be allocated there instead of the in-use chunk's data, effectively replacing the chunk's contents. This means we're able to arbitrarily replace the contents of any in-use chunk.

As we're interested in achieving full code execution, it would be advantageous to locate and overwrite a function pointer in the heap. But… Where can we expect to find such values on the heap? Well, for starters, there are some events in the Wi-Fi standards that must be handled periodically, such as performing scans for adjacent networks. It would probably be a safe bet to assume that the firmware supports handling such periodic timers by using a common API.

Since timers may be created during the firmware's operation, their data structures (e.g., which function to execute and when) must be stored on the heap. To locate these timers, we can reverse-engineer the IRQ vector table entry, and search for the logic corresponding to handling a timer interrupt. After doing so, we find a linked list of entries whose contents seem to conform to that of brcms_timer structure, used in the brcmsmac (SoftMAC) driver. After writing a short script, we can dump the list of timers given a RAM snapshot:

We can see that the timer list is ordered by the timeout value, and most of the timers have a relatively short timeout. Moreover, all the timers are allocated during the firmware's initialisation, and are therefore stored at constant addresses. This is important since, if we'd like to target our free chunk at a timer, we'd need to know it's exact location in memory.

So all that's left is to use our two primitives to replace the contents of one of the timers above with our own data, consequently pointing the timer's function at an address of our choosing.

Here's the game plan. First, we'll use the techniques described above to create two overlapping free chunks. Now, we can use the controlled allocation primitive to point the smaller free chunk at one of the timers in the list above. Next, we create another controlled allocation (freeing the old one). This one will be of size 0x3C, for which the timer chunk is the best-fitting. Therefore, at this point, we'll overwrite the timer's contents.

But which function do we point our timer to? Well, we can use the same trick to commandeer another in-use chunk on the heap, and overwrite its contents with our own shellcode. After briefly searching the heap, we come across a large chunk which simply contains console data during the chip's boot sequence, and is then left allocated but unused. Not only is the allocation is fairly large (0x400 bytes), but it is also placed at a constant address (since it is allocated during the firmware's initialisation sequence) - perfect for our exploit.

Lastly, how can we be sure that the contents of the heap is even executable? After all, the ARM Cortex R4 has a Memory Protection Unit (MPU). Unlike an MMU, it does not allow for the facilitation of a virtual address space, but it does allow control over the access permissions of different memory ranges in RAM. Using the MPU, the heap could (and should) be marked as RW and non-executable.

By reversing the firmware's initialisation routines in the binary, we can see that the MPU is indeed being activated during boot. But what are the contents with which it's configured? We can find out by writing a small assembly stub to dump out the contents of the MPU:

0x00000000 - 0x10000000

AP: 3 - Full access

XN: 0

0x10000000 - 0x20000000

AP: 3 - Full access

XN: 0

0x20000000 - 0x40000000

AP: 3 - Full access

XN: 0

0x40000000 - 0x80000000

AP: 3 - Full access

XN: 0

0x80000000 - 0x100000000

AP: 3 - Full access

XN: 0

Ah-ha - while the MPU is initialised, it is effectively set to mark all of memory as RWX, making it useless. This saves us some hassle… We can conveniently execute our code directly from the heap.

So, at long last, we have an exploit ready! Putting it all together we can now hijack a code chunk to store our shellcode, then hijack a timer to point it at our stored shellcode. Once the timer expires, our code will be executed on the firmware!

At long last, we've gone through the entire process of researching the platform, discovering a vulnerability and writing a full-fledged exploit. Although this post is relatively long, there are many smaller details that I left out in favour of brevity. If you have any specific questions, please let me know. You can find the full exploit, including instructions, here. The exploit includes a relatively benign shellcode, which simply writes a magic value to address 0x200000 in the firmware's RAM, signalling successful execution.

Wrapping Up

We've seen that while the firmware implementation on the Wi-Fi SoC is incredibly complex, it still lags behind in terms of security. Specifically, it lacks all basic exploit mitigations - including stack cookies, safe unlinking and access permission protection (by means of an MPU).

Broadcom have informed me that newer versions of the SoC utilise the MPU, along with several additional hardware security mechanisms. This is an interesting development and a step in the right direction. They are also considering implementing exploit mitigations in future firmware versions.

In the next blog post, we'll see how we can use our assumed control of the Wi-Fi SoC in order to further escalate our privileges into the application processor, taking over the host's operating system!
