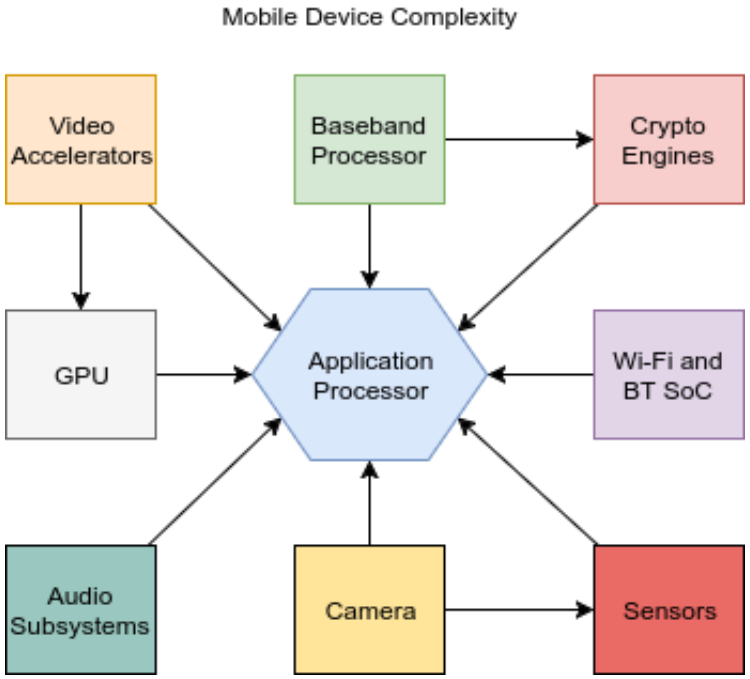
干货丨威胁所有iOS、Android WiFi系统芯片漏洞详解

2017-04-14 行长叠报

众所周知，平台安全是系统安全的组成部分。对于移动设备而言更是如此。如今的移动设备由多个紧密连结的组件构成。在应用处理器（AP）上运行的代码被重点关注，而其它组件的受重视程度则大大削减。



应用处理器所运行的代码，由于安全人员的重点关注，防御力大大增强。以Android为例，包括操作系统、应用程序等的安全性都获得了极大的增强。这些改进的积极性毋庸置疑。然而，攻击者显然更加倾向于寻找阻力最小的入侵路径。提高某一个组件的安全性将不可避免地导致攻击者将注意力转移到别处，从而寻找其它更容易的切入点。

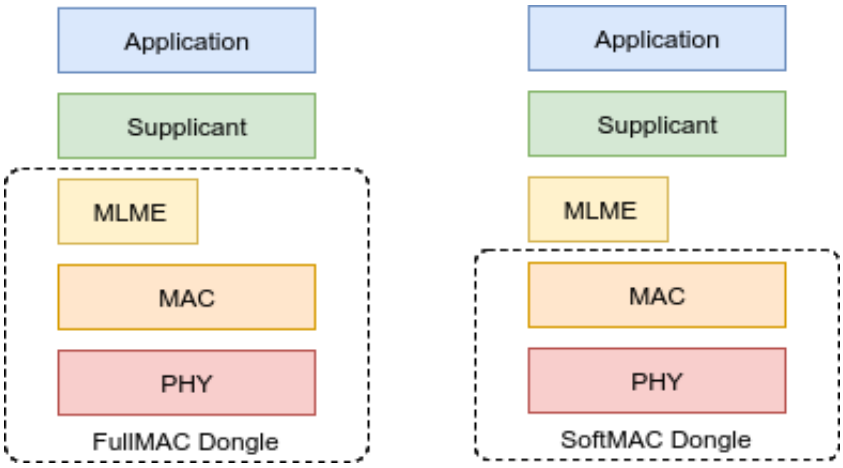
本博客将分为前后两篇，我们将探讨Broadcom在移动设备上使用的WiFi SoC暴露的安全问题。具体来讲，我们的注意力会集中在Android设备上，当然这项研究也适用于应用此WiFi SoC的其它系统。
文章的前篇，将专注于探讨SoC本身，我们将发现和利用允许在芯片上进行远程代码执行的漏洞。
文章的后篇，将进一步探讨如何将权限从SoC提升到系统内核。
文章前后两篇，整体上我们将演示如何在不需用户交互的情况下，通过WiFi接口控制整个设备。

之所以选择Broadcom的WiFi SOC，是因为它们是移动设备上最常用的WiFi芯片组。搭载此芯片的设备包括Nexus5,6和6P，三星大部分旗舰设备以及自iPhone4之后所有型号的iPhone。
本文的演示环境为Android 7.1.1版本的NUF26K，设备型号Nexus 6P。我们将在此环境下演示代码执行漏洞的利用。

为什么选择WiFi

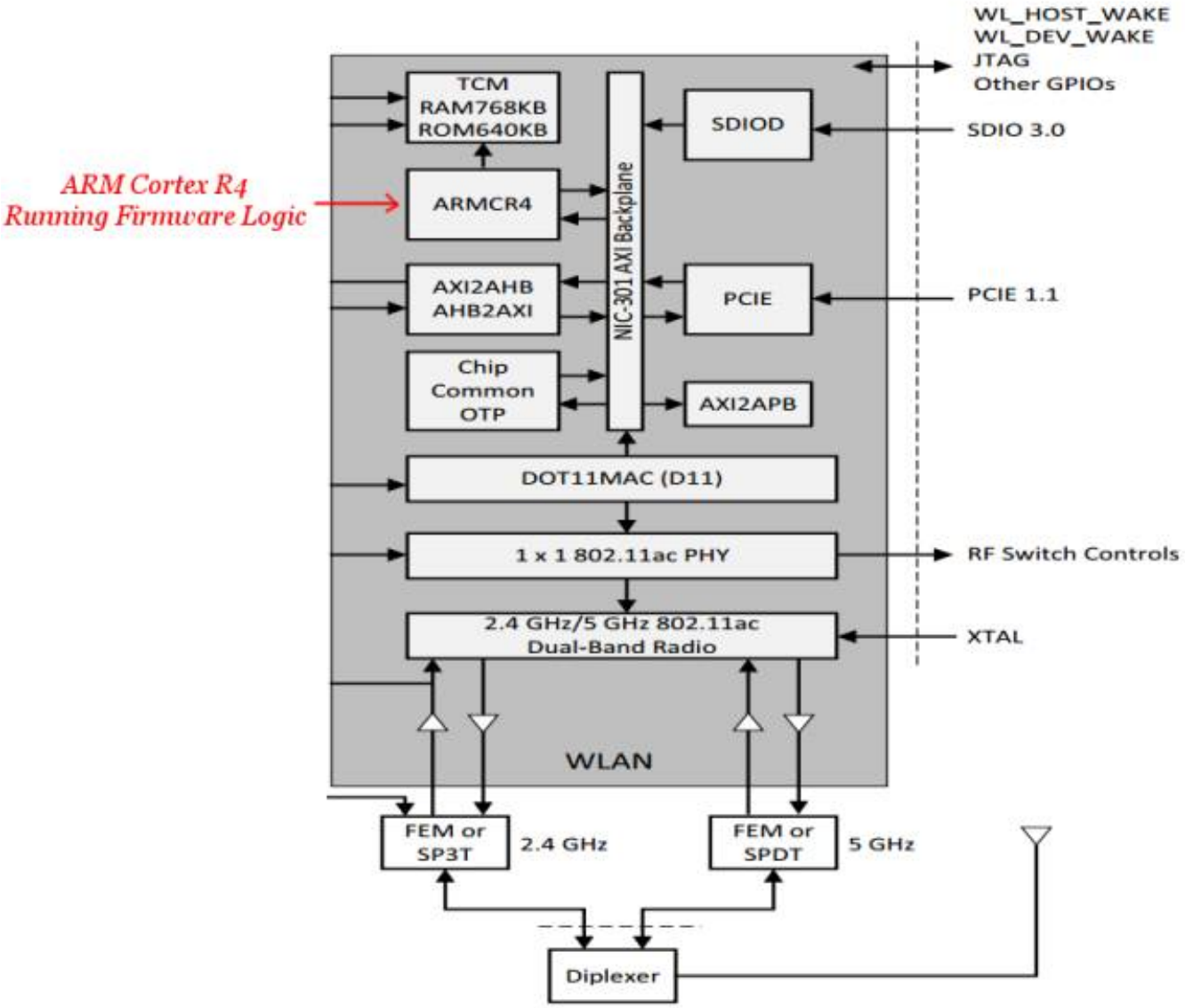
十年以来，WiFi的使用在移动设备上变得相当普遍。WiFi也逐渐演化出了一些难以突破的（安全）设计——一部分着眼于物理细节层面，另一部分侧重于MAC层。为了适应日益增长的复杂性，供应商也开始生产“FullMAC”WiFi SoC。

WiFi FullMAC芯片的推出改善了移动设备的功耗，因为大部分处理是在低功耗SoC而不是耗电量较大的应用处理器上完成的。 更重要的是，FullMAC芯片要更容易集成，因为它们在固件中实现MLME，从而降低了主机端的复杂性。
这一切都表明，推出Wi-Fi FullMAC芯片并没有成本。 引入这些新的硬件，运行专有和复杂的代码库可能会削弱设备的整体安全性，并引入可能危及整个系统的漏洞。



探索平台

为了开始研究，我们至少需要先了解一下WiFi芯片。幸运的是，我们获得了Broadcom Wi-Fi芯片组相关的数据表（尽管是较早的SoC，BCM4339）。通过阅读数据表，我们可以深入了解WIFi芯片组背后的硬件架构。



具体来说，我们可以看到有一个ARM Cortex R4内核，它负责运行处理和处理帧的所有逻辑。 此外，数据手册显示ARM内核具有用于保存固件代码的640KB ROM，以及用于数据处理（例如堆）的768KB RAM，并将补丁存储到固件代码。

要开始分析在ARM内核上运行的代码，我们需要提取ROM的内容，并找到加载到RAM中的数据。

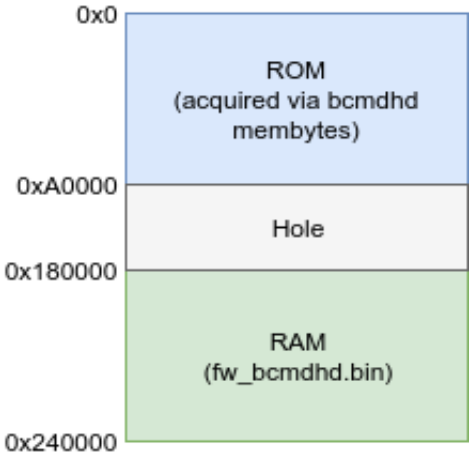
接下来我们要解决第二个问题——ARM内核的RAM中加载了哪些数据？ 这些数据不存在于ROM中，而是芯片首次通电时从外部加载。 因此，通过读取主机驱动程序中的初始化代码，我们应该可以找到包含RAM内容的文件。
实际上，通过驱动程序的代码，我们找到了 BCMDHD_FW_PATH 配置 ，用于表示驱动程序将其内容上传到RAM 的文件的位置。

那么该如何提取ROM的内容?

一种方法是使用主机驱动程序的芯片存储器访问功能（通过SDIO或PCIe上的PIO）直接读取ROM的内容。 但是，这样做将需要修改驱动程序，才能够发出转储ROM所需的命令。另一种方法是将我们自己修改的固件文件加载到RAM中，我们将插入一个可用于转储ROM内存范围的小型存根。
幸运的是，实际上并不需要这些方法。因为 Broadcom提供了一个非常强大的命令行实用程序 dhdutil ， 可用于通过 bcmdhd 驱动程序 与芯片进行交互 。

在该实用程序功能的支持下，我们通过发出特殊命令“membytes”直接读取和写入加密狗（软件保护器）上的内存。 由于我们已经知道了ROM的大小（从数据表中），我们可以直接使用 membytes 命令来读取ROM的内容。
但是，最后一个问题解决之前，首先需要回答 - ROM位于哪里？根据我们的研究，ROM被加载到地址0x0，并且RAM被加载到地址0x180000（而NexMon专注于BCM4339，对于较新的芯片同样适用，如BCM4358）。

集合已经获得的所有信息，我们可以从固件文件中获取RAM的内容，使用dhduil转储ROM，并将这两个文件合并成一个文件，以便我们在IDA中开始分析。

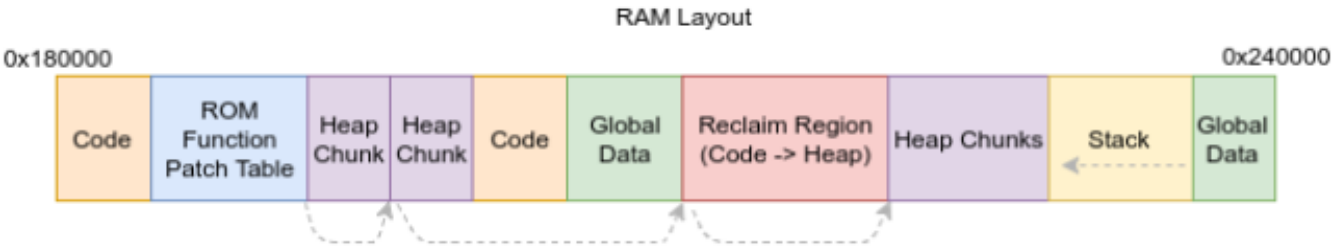


分析固件

由于可用内存（ROM和RAM都相对较小），Broadcom为了节省内存付出了极大的努力。他们删除了二进制文件中的符号和大部分字符串。这增加逆向工程固件代码的难度。此外，他们还选择了专门用于Thumb-2的指令集以实现更好的代码密度。因此，BCM4358上的ROM映像看起来封装紧密——未使用的字节不到300。

但是，这仍然不够。RAM必须适应堆，堆栈和全局数据结构，以及ROM功能的所有补丁或修改。Broadcom决定将固件初始化期间使用的所有功能放在两个特殊区域。初始化完成后，这些区域将被“回收”，然后转换为堆块。

此外，堆块散布在RAM中的代码和数据结构之间，因为后者有时具有对齐要求（或直接从ROM引用，因此无法移动）。 最终的结果是RAM是一堆混乱的堆块，代码和数据结构。



在花了一些时间分析固件后，我们至少可以开始识别一些包含功能名称和其他提示的字符串，帮助我们掌握代码库。 此外，NexMon研究人员提供对应于BCM4339固件的符号表 。 我们可以将相同的符号应用于BCM4339的固件，然后使用bindiff 将更新版本的符号名称与更新的芯片相关联。

方法当然不止此一种——除了我们正在分析的FullMAC SoC之外，Broadcom还生产SoftMAC芯片。由于这些SoftMAC芯片不处理MLME层，所以相应的驱动程序必须执行该处理。 因此，许多Broadcom的MLME处理代码都包含在开源SoftMAC驱动程序 - brcmsmac中 。 虽然这并不能帮助我们了解任何芯片特定的功能或内部处理代码，但它与固件代码共享许多实用功能。

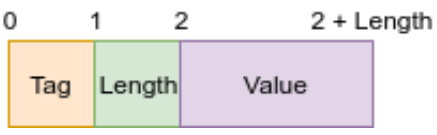
寻找BUG

现在我们掌握了固件的结构，并且有了分析的手段，我们终于可以开始寻找bug了。 但是我们应该从哪里开始?

一种可能的方法是测试固件，以便跟踪在接收和处理数据包时所采用的代码路径。 Cortex R4确实有调试寄存器 ，可用于放置断点并检查各个位置的代码流。

或者，我们可以手动定位一组用于从接收到的帧解析和检索信息的函数，并从那里向后退一步。

这是Wi-Fi常规的处理模式：Wi-Fi管理帧以小的“标记”数据块（称为 信息 资源（IE）） 对大多数信息进行编码 。 这些标记的数据块被构造为 TLV ，其中标签和字段都是单字节。



由于在Wi-Fi帧（除了数据本身之外）传输的大部分信息是使用IE进行编码的，所以它们为我们可以向后退出的良好候选者进行编码。 此外，由于“标签”值是 独一无二的并且是标准化的 ，所以我们可以使用它们的值来帮助我们了解当前处理的代码流。

看看 brcmsmac 驱动程序，我们可以看到Broadcom使用一个功能来从框架中提取IE - bcm_parse_tlvs 。 经过简短搜索（通过关联附近的字符串提示），我们在固件的ROM中找到相同的功能。

现在我们可以开始交叉引用这个调用这个功能的位置，并且将这些调用站点的每一个都相反。 虽然比翻转固件的每一部分要容易得多，但这仍然需要相当长的时间（因为该功能具有超过110个交叉引用，有些涉及其他包装器功能，这些功能本身是从多个位置调用的）。

进行逆向工程后，我发现了一个很少在管理框架中嵌入的信息元素的处理相关的漏洞。

当设备连接上支持无线漫游功能的网络时，可以触发两个漏洞：
802.11r F ast BSS T 转换（FT）或Cisco的CCKM漫游。

这两个漏洞的利用方式相对直接——他们是最简单的堆栈溢出。并且，在固件（指NHDRTE）上运行的操作系统不使用堆栈cookie，因此不需要额外的信息泄露或旁路。

然而，虽然原理相对简单，但是想利用漏洞还是需要一些设置。首先，我们需要一个发射支持这些功能的WiFi网络信号。802.11rFT是基于hostapd实现的开放标准。

另外，问需要弄清楚哪些设备支持上述功能。Broadcom提供许多由客户授权的功能——但并非所有的设备上都存在所有功能（实际情况是，相应的补丁可能根本不适合RAM）

幸运的是，Broadcom可以很容易地区分每个固件映像中实际存在哪些功能。 下载到芯片的RAM内容中的最后几个字节包含固件的“版本字符串”。 此字符串包含固件编译的日期，芯片的修订版本，固件版本以及破折号“标签”列表。 每个标签表示固件映像支持的功能。 例如，以下是Nexus 6P的版本字符串：
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-d11statusVersion: 7.112.201.1 (r659325) CRC: 8c7aa795 Date: Tue 2016-09-13 15:05:58 PDTUcode Ver: 963.317 FWID: 01-ba83502b

802.11r FT功能的存在由“fbt”标签指示。 类似地，CCKM的支持由“ccx”标签指示。 不幸的是，Nexus 6P似乎并不支持这些功能。 事实上，在我自己的Android固件镜像库中快速搜索“ccx”功能（CCKM支持），这表明Nexus设备不支持此功能，但 支持各种各样的三星旗舰设备，其部分列表包括Galaxy S7（G930F，G930V），Galaxy S7 Edge（G935F，G9350），Galaxy S6 Edge（G925V）等。

那么其他漏洞呢？ 它们都涉及隧道直接链接设置（TDLS）的实现。 TDLS连接允许Wi-Fi网络上的对等设备在彼此之间交换数据，而不通过AP 接入点（AP），从而防止AP拥塞。

固件中对TDLS的支持由“betdls”和“tdls”标签指示。 通过我的固件存储库搜索，我可以看到绝大多数设备确实支持TDLS。 这包括所有最近的Nexus设备（Nexus 5,6,6P）和大多数三星旗舰产品。

此外，TDLS被指定为 802.11z标准的 一部分 （需要IEEE订阅）。 由于有关TDLS的所有信息都可用，我们可以阅读该标准，以便熟悉Broadcom实施中的相关代码路径。 作为开放标准，它还受到开源请求者的支持，
如 wpa_supplicant 。 因此，我们可以检查wpa_supplicant中的TDLS功能的 实现 ，以进一步提高对固件中相关代码的理解。

最后，正如我们稍后将看到的，触发这两个漏洞可以由Wi-Fi网络上的任何对等设备完成，而不需要对设备进行物理交互。

原理

为了确保在安装和拆卸阶段期间传输的消息的完整性，相应的TDLS帧包
括 Message Integrity Codes （MIC）。对于设置阶段，一旦接收到第二握手消息（M2），TPK可以由双方派生。使用TPK，TDLS发起者可以计算第三个握手帧内容的MIC，然后由TDLS发起者进行验证。

在握手帧中编码的IE的内容上计算MIC，如下所示：

$$AES-CMAC_{TPK-KCK}\left(\begin{array}{ccccc} InitiatorMAC & || & ResponderMAC & || \\ TransactionSeq & || & Link\,ID-IE & || \\ RSN-IE & || & Timeout\,Interval-IE & || \\ Fast\,Transition-IE & & & \end{array}\right)$$

类似地，拆卸框架还包括通过 略微不同的IE集合计算 的MIC：

$$AES-CMAC_{TPK-KCK}\left(\begin{array}{ccccc} Link\,ID-IE & || & ReasonCode & || \\ DialogToken & || & TransactionSeq & || \\ Fast\,Transition-IE & & & \end{array}\right)$$

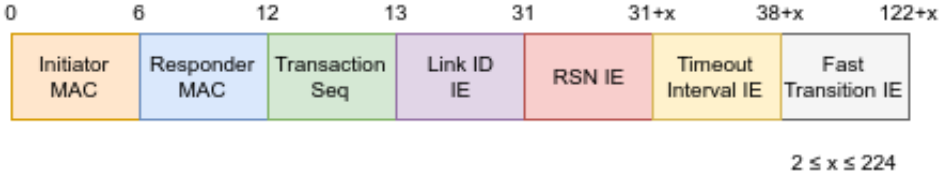
那么我们如何在固件的代码中找到这些计算呢？ 那么运气好的话，一些指向TDLS的字符串被遗留在固件的ROM中，这样我们可以快速回到相关的功能。
在进行TDLS动作帧处理后的大部分流程逆向工程后，我们终于达到负责处理TDLS Setup Confirm（PMK M3）帧的功能。 该函数首先执行一些验证，以确保请求是合法的。 它查询内部数据结构，以确保TDLS连接确实正在与请求对等体建立。 然后，它验证Link-ID IE（通过检查其编码的BSSID与当前网络的匹配），并且还验证32字节的启动器随机数（“Snonce”）值（通过将其与存储的初始随机数进行比较）。

一旦建立了一定程度的置信度，请求可能确实是合法的，则该功能将继续调用内部帮助函数，任务是计算MIC并确保它与帧中编码的协议一致。 非常有帮助，固件还包括此功能的名称（“ wlc_tdls_cal_mic_chk ”）。
在逆功能化功能之后，我们得到以下近似的高级逻辑：

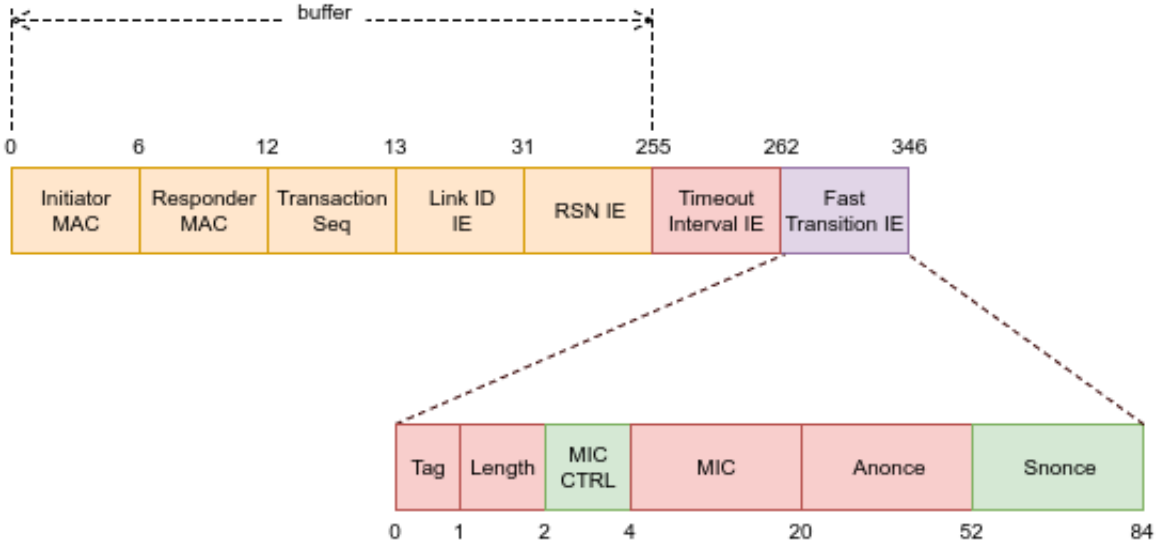
```
uint8_t* buffer = malloc (256);
uint8_t * pos = buffer ;
//复制初始（静态）信息
uint8_t * linkid_ie = bcm_parse_tlvs (...，101);
memcpy ( pos ， linkid_ie + 0x8，0x6); pos + = 0x6; //启动器MAC
memcpy (pos ， linkid_ie + 0xE，0x6); pos + = 0x6; //响应者MAC
* pos = transaction_seq; pos ++; //TransactionSeq
memcpy ( pos ， linkid_ie，0x14); pos + = 0x14; // LinkID-IE
//复制RSN IE
uint8_t * rsn_ie = bcm_parse_tlvs (...，48);
if(rsn_ie [1] + 2 + （ pos - buffer ） > 0xFF){
//处理溢出
}}
memcpy （ pos ， rsn_ie，rsn_ie [1] + 2）; pos + = rsn_ie [1] + 2; // RSN-IE
//复制剩余的IE
```

```
uint8_t * timeout_ie = bcm_parse_tlvs （...， 56）；
uint8_t * ft_ie = bcm_parse_tlvs （...， 55）；
memcpy （ pos ， timeout_ie， 0x7）; pos + = 0x7; //超时间隔IE
memcpy （ pos ， ft_ie， 0x54）; pos + = 0x54; //快速转换IE
```

如上所述，虽然功能验证了RSN IE的长度不超过分配的缓冲区长度（第13行），但是它无法验证后续的IE也不会溢出缓冲区。 因此，将RSN IE的长度设置为较大的值（例如，使得rsn_ie [1] + 2 +（pos-buffer）== 0xFF）将导致超时间隔和快速转换IE被复制出超出范围，溢出缓冲区。



例如，假设我们将RSN IE（x）的长度设置为其最大可能值224，我们到达以下元素的位置：



在这个图中，橙色域是溢出的“无关紧要”，因为它们位于缓冲区的边界内。 红色字段表示我们无法完全控制的值，绿色字段表示完全可控的值。

FTIE的标签和长度字段是恒定的，因此不可控。 TDSI应答会随机选择32位“Anonce”值，将其置于我们的影响范围之外。

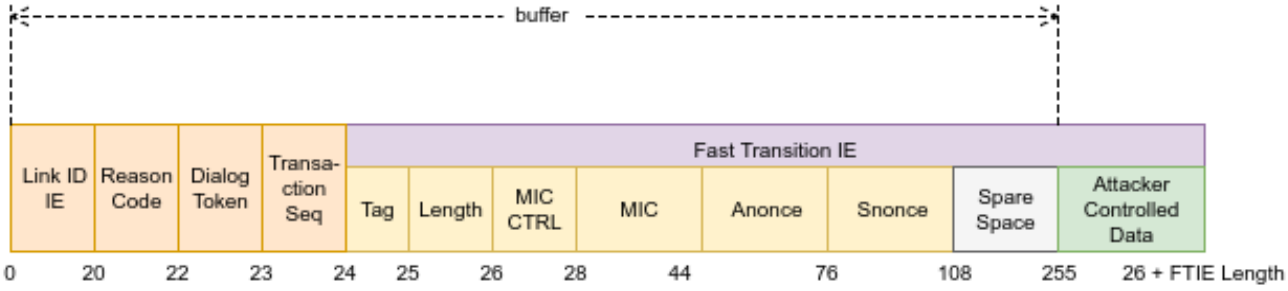
但是，FTIE本身中的几个字段是可以任意选择 - 例如，在获取第一个握手帧时，TLDS发起者选择“Snonce”值。 此外，因为在执行该功能之前未进行验证，所以FTIE中的“MIC控制”字段可以自由选择。

无论如何，现在我们已经对安装阶段的MIC验证进行了审核，在拆卸阶段，我们将目光转向MIC验证。 那里的代码是不是也存在问题?

看看在拆卸阶段
(“ wlc_tdls_cal_mic_chk”) 中的MIC计算 ， 我们得到以下高级逻辑:
uint8_t * buffer = malloc (256);
uint8_t * linkid_ie = bcm_parse_tlvs (..., 101); //链接ID
memcpy(buffer , linkid_ie, 0x14);
uint8_t * ft_ie = bcm_parse_tlvs (..., 55);
memcpy (buffer + 0x18, ft_ie, ft_ie [1] +2); //快速转换IE

显然，这又是一次直接溢出。 FT-IE的长度字段未被验证，以确保它不超过分配的缓冲区的长度。
这意味着只要精心炮制的FT-IE，就可以触发溢出。
当然，在触发易受攻击的代码路径之前，依然存在验证，以限制了对溢出元素的控制。

我们来试试绘制元素溢出的位置:



很显然，我们不需要担心在溢出之前验证的FTIE中存储的值，因为它们全部放置在缓冲区的范围内。相反，攻击者控制的部分只是不需要进行任何验证的备用数据，因此可以由我们自由选择。 也就是说，溢出的 程度 是非常有限的 - 我们只能覆盖 超过缓冲区范围的 最多 25个字节。

漏洞脚本

调查堆状态

如今我们已经掌握了基本的资料。是时候来测试我们的假设是否符合现实。

为了做到这一点，我们需要一个测试台，使我们能够发送制作的帧，触发溢出。

我们使用 wpa_supplicant 作为基础制作我们的框架。 这样可以帮助我们重现和维护TDLS连接所需的所有逻辑。

为了测试这些漏洞，我们将修改 wpa_supplicant ，让我们发送包含过大FTIE的TDLS拆分帧。 通过 wpa_supplicant 的代码，我们可以快速识别负责生成和发送拆卸框架的功能 - wpa_tdls_send_teardown 。 通过对此函数（绿色）添加一些小的更改，我们应该能够在接收到触发器的情况下触发溢出，从而导致25个字节的0xAB被写入OOB：
staticint 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）; //用0xAB溢出

 os_memcpy （ ftie -> Anonce，peer-> rnonce，WPA_NONCE_LEN）；
 os_memcpy （ ftie -> Snonce，peer-> inonce，WPA_NONCE_LEN）；
 pos + = ftie - >ie_len + 2;
 ...
}

现在，我们只需要与 wpa_supplicant 进行交互， 就 可以建立和拆除与目标设备的TDLS连接。 wpa_supplicant 支持许多命令接口，包括一个名为 wpa_cli 的命令行实用程序 。 此命令行界面还支持其它的TDLS功能的命令：
TDLS_DISCOVER - 发送“TDLS发现请求”帧并列出响应

TDLS_SETUP - 创建与给定MAC地址的对等体的TDLS连接

TDLS_TEARDOWN - 使用给定的MAC撕开与对等体的TDLS连接

实际上，在 使用TDLS支持（CONFIG_TDLS） 编译 wpa_supplicant 之后 ， 建立一个网络，并将我们的目标设备和研究平台连接到网络上。

```
wlp3s0: Control interface command 'TDLS_DISCOVER 24:DF:6A:CF:6F:09'
CTRL_IFACE TDLS_DISCOVER 24:df:6a:cf:6f:09
TDLS: Sending Discovery Request to peer 24:df:6a:cf:6f:09
TDLS: TPK send dest=24:df:6a:cf:6f:09 action_code=10 dialog_token=1 status_code=0 peer_capab=0 initiator=
1 msg_len=0
CTRL-DEBUG: ctrl_sock-sendto: sock=13 sndbuf=212992 outq=0 send_len=3
nl80211: Event message available
nl80211: BSS Event 59 (NL80211_CMD_FRAME) received for wlp3s0
nl80211: MLME event 59 (NL80211_CMD_FRAME) on wlp3s0(5c:e0:c5:8b:da:ee) A1=5c:e0:c5:8b:da:ee A2=24:df:6a:
cf:6f:09
nl80211: MLME event frame - hexdump(len=136): d0 08 3a 01 5c e0 c5 8b da ee 24 df 6a cf 6f 09 90 13 37 13
 37 10 90 15 04 0e 01 31 04 01 08 82 84 8b 96 24 30 48 6c 32 04 0c 12 18 60 24 02 01 0b 30 14 01 00 00 0f
 ac 07 01 00 00 0f ac 04 01 00 00 0f ac 07 00 02 7f 08 c8 06 18 50 ba a7 39 05 38 05 02 e0 93 04 00 2d 1a
 2d 00 17 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 65 12 90 13 37 13 37 10 5c
 e0 c5 8b da ee 24 df 6a cf 6f 09
nl80211: Frame event
nl80211: RX frame da=5c:e0:c5:8b:da:ee sa=24:df:6a:cf:6f:09 bssid=90:13:37:13:37:10 freq=2422 ssi_signal=
-18 fc=0x8d0 seq_ctrl=0x1590 stype=13 (WLAN_FC_STYPE_ACTION) len=136
wlp3s0: Event RX_MGMT (19) received
wlp3s0: Received Action frame: SA=24:df:6a:cf:6f:09 Category=4 DataLen=111 freq=2422 MHz
wlp3s0: TDLS: Received Discovery Response from 24:df:6a:cf:6f:09
```

我们现在可以发送一个TDLS_SETUP命令，其次是我们制作的TDLS_TEARDOWN。 如果一切正常，应该会触发溢出。

然而，让我们意识到一个尴尬的问题——我们将如何 知道 溢出发生时间?

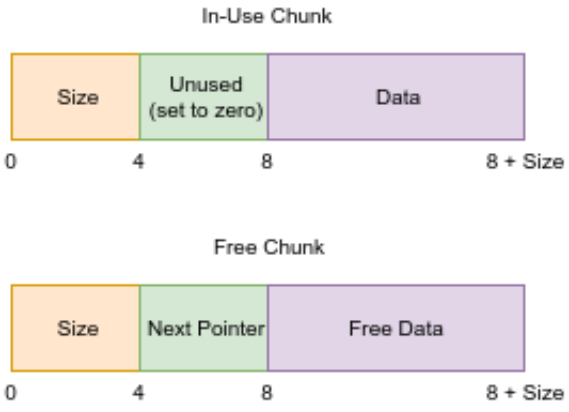
有可能我们溢出的指令并未被执行。

也有可能，在我们毫无觉察的情况下执行使固件设备崩溃。

要搞清楚这个问题，我们需要了解Broadcom堆背后的逻辑。

实际上我们验证发现逻辑非常简单—— 它会自动分配“最合适”的区域，自动保留一个单独链接的空闲块列表。

当分块分配时，它们是从最适合的自由块（最大的块，足够大的）的最后（最高地址）选择。 堆块具有以下结构：



通过逆向分配器的逻辑，我们可以找到指向RAM中第一个空闲块头部的指针的位置。 因此我们可以创建一个实用程序，以固件的RAM的转储，绘制堆的freelist的当前状态。 通过使用 dhdutil 的“upload”命令 可以很容易地获取固件的RAM快照 。

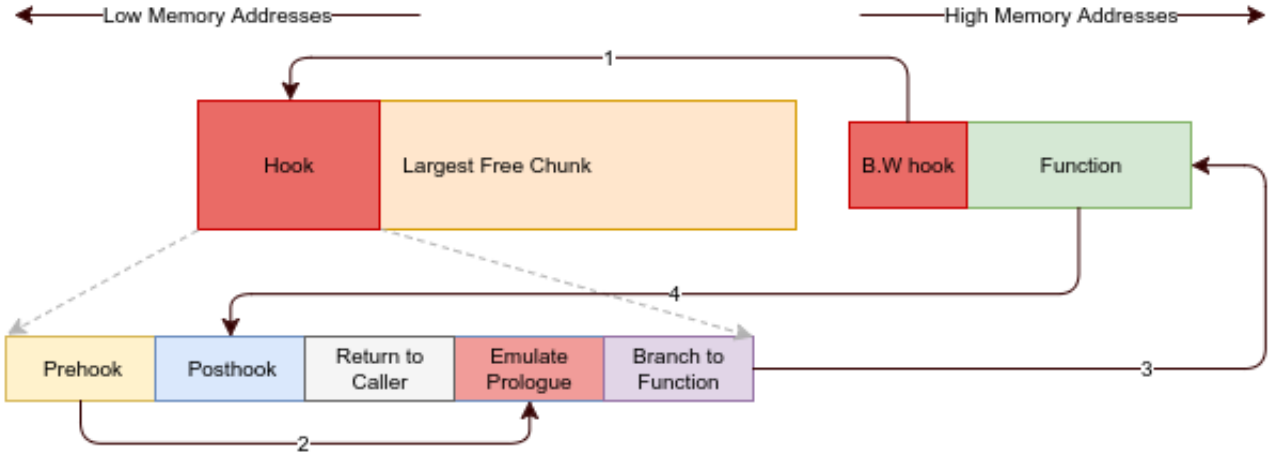
在写了一个小的可视化脚本后，遍历堆的freelist并将其内容导出为 dot ， 我们可以使用graphviz绘制freelist的状态，如下所示：

现在，我们可以发出制作的TDLS_TEARDOWN框架，立即拍摄固件RAM的快照，并检查freelist是否有任何变化的迹象：

由于分配器默认“最适合”的选择，所以只要存在其他足够大的空闲块，后续分配将不会被放置在此块中。 这也意味着固件不会崩溃，而是继续正常运行。

现在我们已经确认了溢出事实上已经发生了，可以进入下个阶段。
为了让我们能够在安装和拆卸过程中监视堆的状态，还需要一些简单的工具。方便我们追踪固件中malloc和free函数的参数和返回值 。

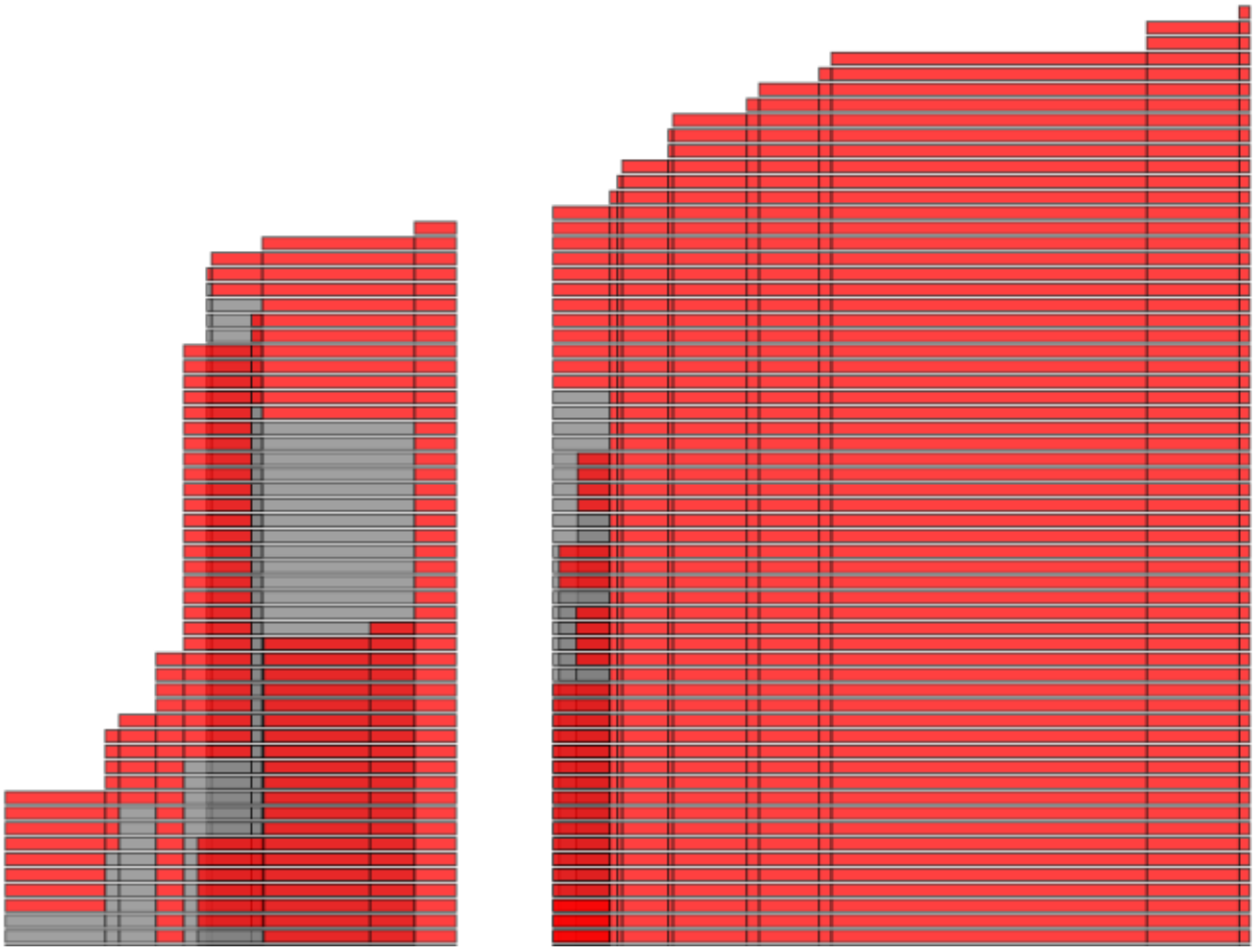
重要的是要注意， malloc 和 free 函数都存在于RAM中（它们是RAM的代码块中的第一个函数）。 这方便我们自由地重写他们的序言。
简而言之，修补程序是相当标准的 - 它将补丁的代码写入RAM中的一个未使用的区域（堆中最大的空闲块的头），然后插入一个Thumb-2宽分支（这恰巧是也许是最简单的编码，我见过的操作码 - 见 4.6.12 T4 ）从挂钩函数的序言到钩子本身。



使用我们的新修补程序，我们现在可以调用 malloc 和 free 函数，以添加跟踪，使我们能够跟踪堆上发生的每个操作。 然后可以通过发出 dhdutil 来从固件的控制台缓冲区中读取这些踪迹 “consoledump”命令。请注意，在一些较新的芯片上，如Nexus 6P上的BCM4358，此命令失败。这是因为Broadcom忘记 在指向控制台的数据结构的固件 中添加 魔术指针 的偏移量 。您可以通过向驱动程序添加正确的偏移量（ 请参阅 debug_info_ptrs ）或将值和指针写入列表中的探测内存地址之一来解决此问题。

使用新获取的轨迹，我们可以编写一个更好的可视化程序，使我们能够在整个设置和拆

卸阶段跟踪堆的状态。这种可视化器将可以看到堆上发生的每个操作，提供更细粒度的数据。不用多说，我们来看看建立TDLS连接时的堆活动：



纵轴表示时间 - 在 malloc 或 free 操作 后，每行都是新的堆状态 。横轴表示空格 - 左侧的较低地址，而右侧的地址较高。 红色块表示正在使用的块，灰色块表示空闲块。

如上所述，建立TDLS连接是一个 凌乱的 过程。有大量和小型的地区有很多的分配和释放。此外，溢出发生在设置阶段所以不允许我们在触发溢出之前对堆的状态进行很多的控制。

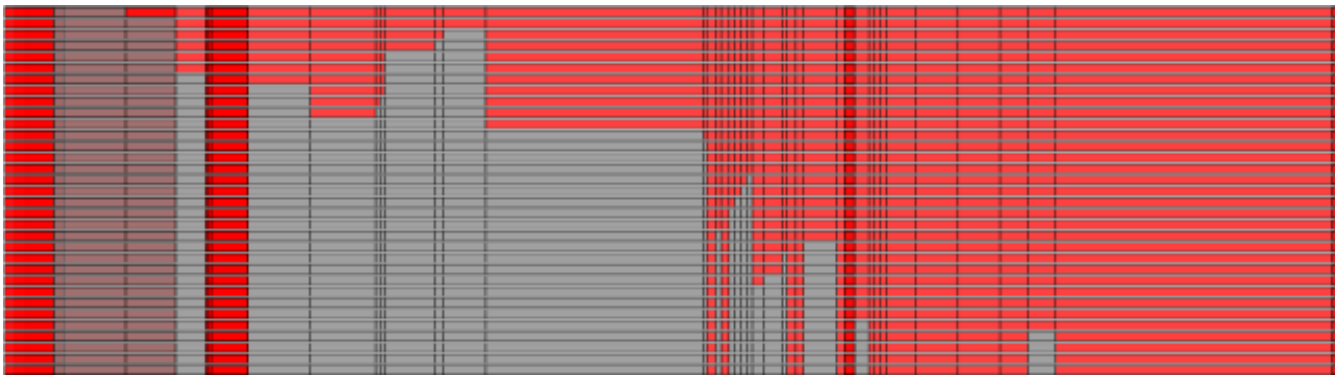
然而，退一步，我们可以观察到一个相当令人惊讶的事实。除了在TDLS连接建立期间的堆活动，其余时间堆上几乎没有任何活动。
事实上发送和接收的帧是共享池进行而不是在堆中进行的。不仅如此，但是它们的处理不会导致 单 堆操作——一切都在“就地”完成。即使尝试通过发送包含异常位组合的随机帧来有意地导致分配，固件的堆仍然在很大程度上不受影响。

这对我们来说有利有弊。一方面，这意味着堆的结构是高度一致的。在数据结构分配很

少的事件中，它们立即释放，并将堆恢复到原始状态。

另一方面，这意味着我们 对堆结构的 控制 程度 相当有限 。在大多数情况下，在固件初始化之后，堆的任何结构都是我们将要使用的（除非我们找到一些原始的，将允许我们更好地塑造堆）。
也许我们应该看看拆卸阶段呢?

虽然这些深入的踪迹对于获取堆状态的高级视图很有用，但是它们很难被破译。事实上，在大多数情况下，只需要对堆的单个快照进行可视化即可，就像我们之前使用graphviz可视化器一样。在这种情况下，允许它根据堆的单个快照生成详细的图形输出来改进我们以前的堆可视化。



正如我们之前看到的，我们可以“访问”freelist来获知每个空闲块的位置和大小。 此外，我们可以通过访问 自由块之间 的间隙，并通过读取每个使用块的字段“大小”来推断该使用块的位置 。我已经写了另一个可视化器，它只是通过一系列“快照”图像生成堆状态的可视化。

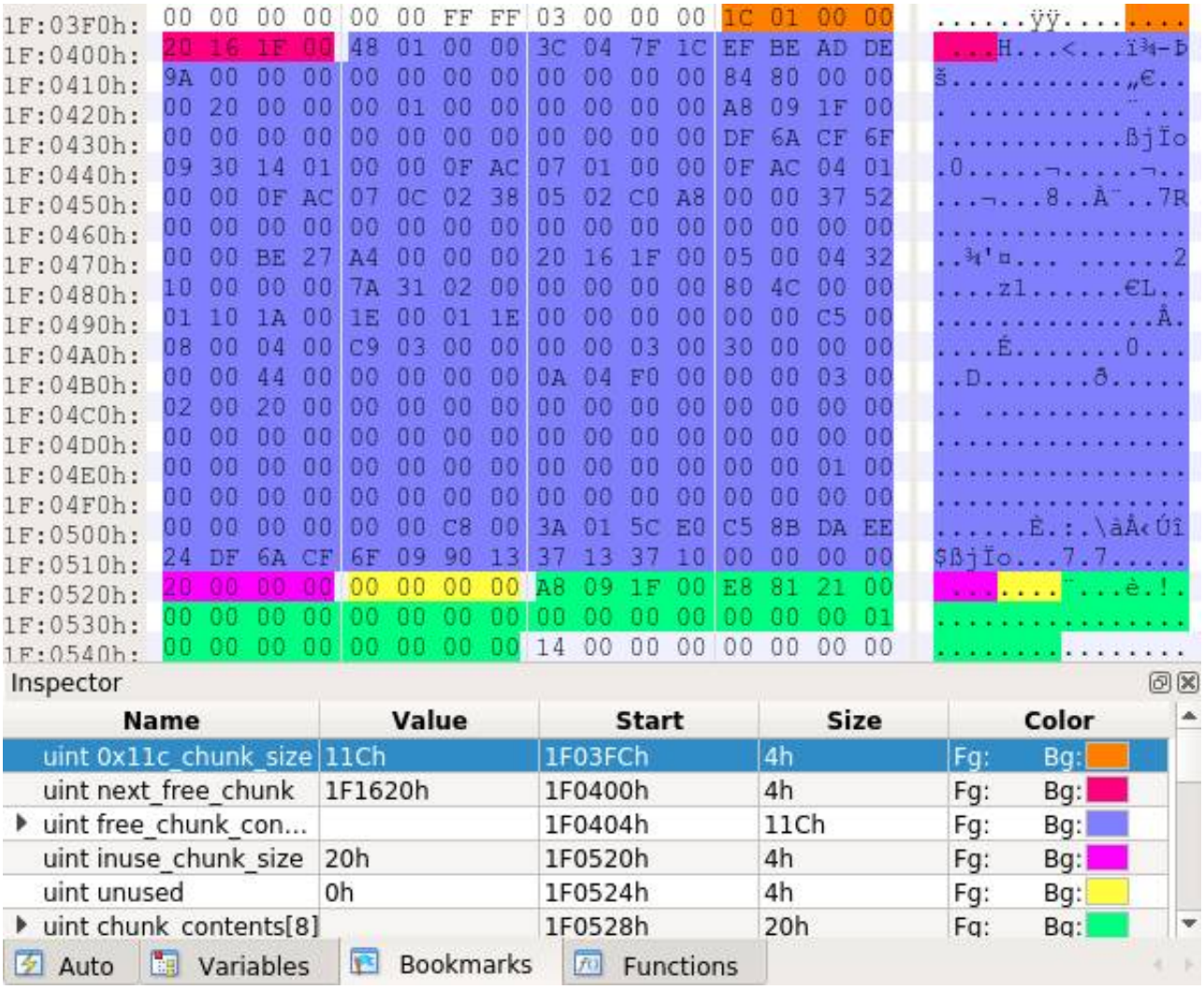
使用此可视化器，现在可以在设置TDLS连接后查看堆的状态 。当我们在拆卸阶段触发溢出时，这将是我们需要处理的堆状态。

我们可以看到，在建立TDLS连接之后，大多数堆的二进制块是连续的，但是也形成了两个孔; 大小为0x11C，另一个大小为0x124。 激活拆卸阶段的跟踪，我们可以看到发生以下分配：

（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

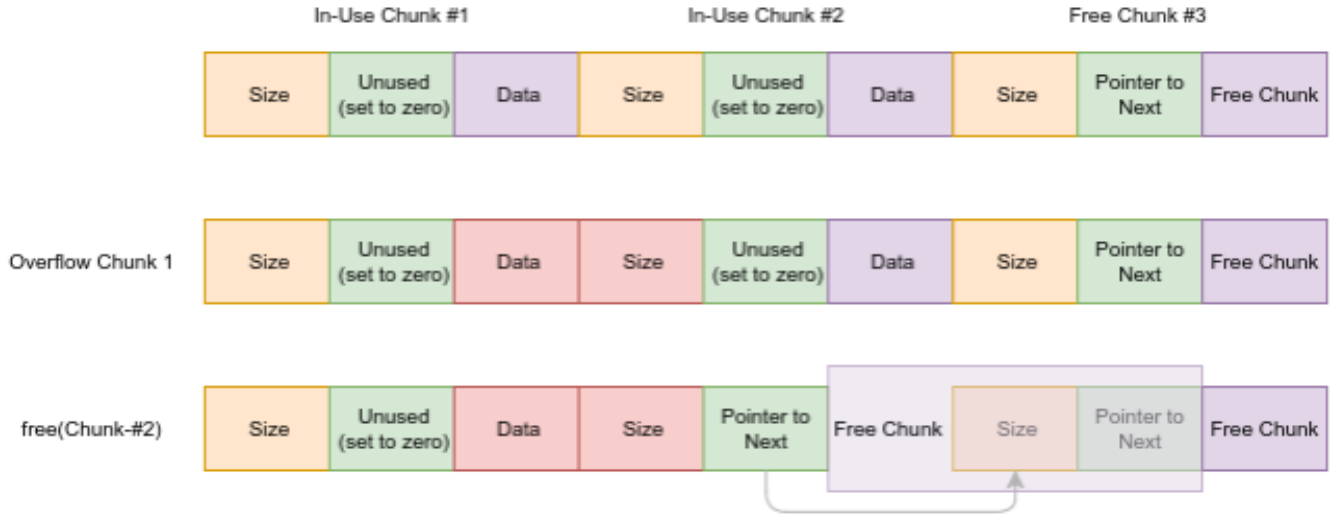
（34）free-ptr： 1f2a44
（35）free-ptr： 1f1654
（36）free-ptr： 1f0480
（37）malloc - size：256，caller：7aa15，res： 1f0420
（38）malloc - size：16，caller：7aa23，res： 1f1658

突出显示的行表示对于拆卸框架的MIC计算的256字节缓冲区的分配，我们可以使用我们的漏洞溢出相同的缓冲区。 此外，在发送溢出帧之前，似乎堆活动相当低。 将上面的堆快照与跟踪文件组合，我们可以推断出256字节缓冲区中最适合的块在0x11C字节的孔中。 这意味着使用我们的25字节溢出，我们可以覆盖：
下一个使用中的块的标题
从下一个使用块的内容开始几个字节
我们来仔细看看下一个使用中的大块，看看有没有什么有趣的信息，我们想覆盖那里：



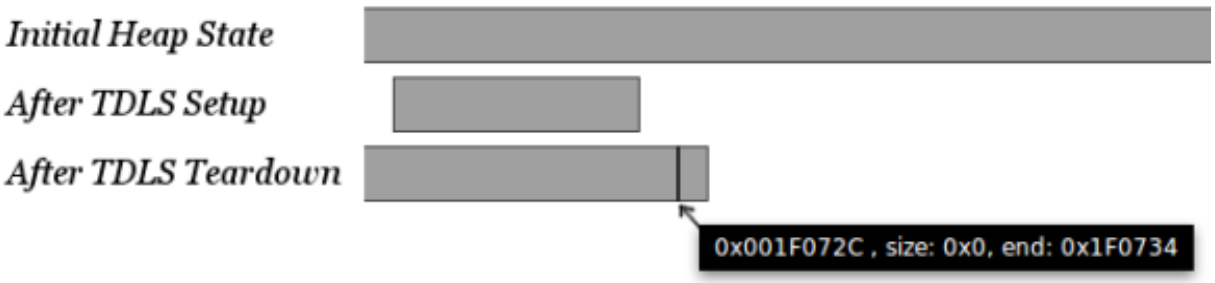
所以下一个大块大部分是空的，除了靠近它的头几个指针。我们可以通过手动破坏这些指针（将它们指向无效的存储器地址，例如0xCDCDCDCD），以及检测固件的异常向

量来查看是否崩溃。 但尝试过后否定了这种可能。
因此我们想到了另外一种可能性——破坏使用中的块的“大小”字段。 回想一下，一旦TDLS连接被拆除，与之相关的数据结构将被释放。 释放大小我们已经损坏的使用中的大块可能会产生许多有趣的后果。
对于初学者来说，如果我们减小块的大小，我们可以有意地“泄漏”缓冲区的尾端，使其永远保持不可分配。 另外，我们可以将块的大小设置为更大的值，从而导致下一个空闲操作创建一个空闲块，其尾端与另一个堆块重叠。



MIC检查只是TDLS连接断开时发生的许多操作之一。一旦在收集TDLS会话的数据结构中被释放，它可能会成为拆卸过程中后续分配的最佳选择。 这些分配可能会导致堆不稳定，甚至会使固件崩溃。

假设我们只对不大于RAM本身的块大小感兴趣，我们可以简单地枚举通过覆盖生成的每个堆状态下一个块的“size”字段具有给定值并拆除连接。 这可以通过在发送（执行枚举）上使用脚本来自动执行，同时在设备上同时获取RAM的“快照”，并观察其状态（无论它们是否一致，以及固件是否被恢复操作拆卸后）。
具体来说，如果我们能够创建一个堆状态，这样两个空闲块就会重叠，这将是非常有利的。 在这种情况下，从一个块获取的分配可以用于损坏另一个空闲块的“下一个”指针。 这也许可以用来控制后续分配的位置。
通过用值72覆盖“size”字段并拆除连接，我们实现了以下堆状态：



在拆除连接之后，我们会留下一个零大小的空闲块，重叠一个不同的（较大的）空闲块！ 这意味着一旦从大块中分配了一个分配，它会损坏较小块的“大小”和“next”字段。 这可能是非常有用的 - 我们可以尝试并将下一个空闲块指向一个内容地址，其内容我们要修改。 只要该地址中的数据符合一个空闲块的格式，我们可能可以控制堆在该地址上覆盖随后分配的内存。

所以在拆除连接之后，我们会留下一个大小为0的空闲块，重叠一个不同的（较大的）空闲块！ 这意味着一旦从大块中分配了一个分配，它会损坏较小块的“大小”和“下一个”字段。 这可能是非常有用的 - 我们可以尝试并将下一个空闲块指向一个内容地址，其内容我们要修改。 只要该地址中的数据符合一个空闲块的格式，我们可能可以说服堆在该地址上覆盖随后分配的内存。

寻找受控分配

为了开始探索这些可能性，我们首先需要创建一个 受控的分配原语， 这意味着我们要么控制分配的大小，要么是内容，要么是（理想的）两者。 回想一下，正如我们以前看到的， 在固件的正常处理 中实际上 很难 触发分配 - 几乎所有的处理都是在现场完成的。 此外，即使 是 分配 数据 的 情况，其寿命也很短; 一旦不再使用内存就立即收回。

就这样，我们已经看到至少有一组数据结构，其生命周期是可控的，并且包含多个不同的信息 - TDLS连接本身 。只要有效，固件必须保留与TDLS连接相关的所有信息。也许我们可以找到一些与TDLS相关的数据结构，可以作为受控分配的好候选者?

要搜索一个，我们先看看处理每个TDLS动作帧的功能
- wlc_tdls_rcv_action_frame 。 该功能从读取TDLS类别和动作代码开始。 然后，根据接收到的动作代码，将帧路由到适当的处理函数。

```
else if ( action_code > 4 )
{
  if ( action_code == 9 )                          // Peer Traffic Response
  {
    wlc_tdls_process_peer_traffic_response(a1, a2, v4, v13);
  }
  else if ( action_code > 9 )
  {
    if ( action_code == 10 )                       // Discovery Request
    {
      wlc_tdls_process_discovery_request(a1, a2, v4, v13);
    }
    else if ( action_code == 127 )                 // What's this???
    {
      wlc_tdls_process_vendor_specific(a1, a2, v4, v13);
    }
  }
  else if ( action_code == 5 )                     // Channel Switch Request
  {
    wlc_tdls_process_chsw_req(a1, a2, v4, v13);
  }
```

我们可以看到，除了常规的规范定义的动作代码之外，固件还支持超出规范的框架，其动作代码为127.任何超出规格的内容都会自动怀疑，因此可能会很好像任何地方寻找我们的原始。

确实，挖掘这个功能，我们发现它会执行以下任务。 首先，它验证帧内容中的前3个字节与Wi-Fi连接 OUI （50：6F：9A） 相匹配 。 然后，它检索帧的第四个字节，并将其用作“命令代码”。 目前，仅实现了两个供应商特定的命令，命令＃4和＃5。 在高层次上 命令＃4用于通过TDLS连接发送隧道式探测请求，命令＃5用于向主机发送“事件”通知，指示“特殊”帧到达。

然而，更有趣的是，我们看到第4号命令的实现与我们目前的追求相似。 首先，它不需要TDLS连接的存在才能被处理！ 这样就可以在拆除连接后发送框架。 其次，通过在此函数执行期间激活堆跟踪并对其逻辑进行逆向工程，我们发现该函数触发下列高级事件序列：

```
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);
```

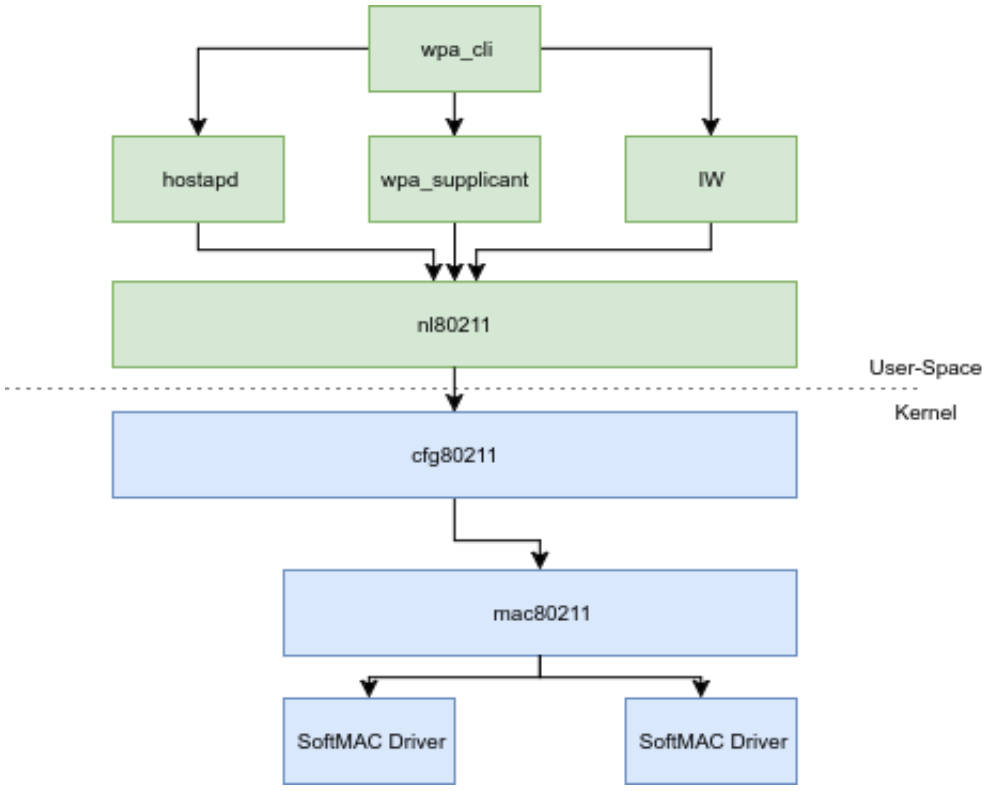
很好！ 所以我们得到一个可控制的生命周期的分配（A），受控的大小和受控的内

容！ 我们还可以做什么？

虽然有一个小小的障碍，修改 wpa_supplicant 发送此精心制作的TDLS帧会导致故障。 虽然 wpa_supplicant 允许我们完全控制TDLS框架中的许多字段，但它只是一个请求者，而不是MLME实现。 这意味着相应的MLME层负责编写和发送实际的TDLS帧。

在我正在使用的攻击平台上，我有一台运行Ubuntu 16.04的笔记本电脑和一个TP-Link TL-WN722N加密狗。加密狗是SoftMAC配置，因此MLME层是Linux内核中存在的，即“cfg80211”配置层。

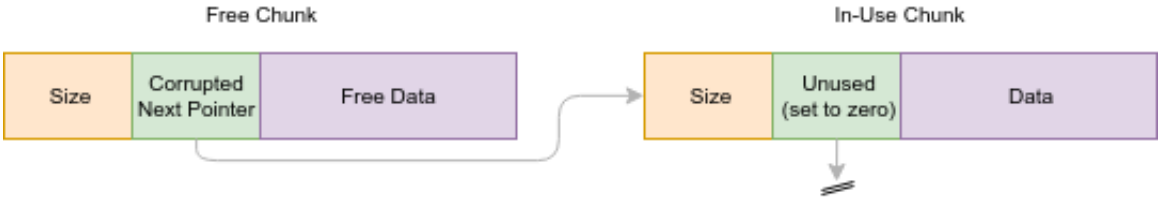
当 wpa_supplicant 希望创建和发送TDLS帧时，它通过 Netlink 发送特殊请求 ，然后由cfg80211框架处理，然后传递给SoftMAC层“mac80211”。 然而， 令人遗憾的是，mac80211 无法处理特殊的供应商框架，只是拒绝它们。 尽管如此，这只是一个小小的不便 - 我已经写 了一些补丁到 mac80211 ，这增加了对这些特殊供应商框架的支持。 应用这些补丁后，重新编译和引导内核，我们现在可以发送我们制作的框架。



为了更容易地控制供应商框架，我还在 wpa_supplicant 的CLI - “TDLS_VNDR”中 添加了对新命令的支持 。该命令允许我们将任意数据的TDLS供应商帧发送到任何MAC地址（无论是否建立到该对等体的TDLS连接）。

在创建两个重叠的块之后，我们现在可以使用我们的受控分配原语从较大块的尾部分配

内存，从而将较小的空闲块指向我们选择的位置。 但是，无论我们选择哪个位置，都必须对“size”和“next”字段都有有效的值，否则后来的 malloc 和 free 可能会失败，可能会导致固件崩溃。
回想一下，使用中的块在相同的位置指定他们的大小字段，它们的空闲块。对于“next”指针，它在空闲块中未使用，但 在分配块期间设置为零 。 这意味着通过破坏自由列表来指向使用中的块，我们可以欺骗堆，认为它只是另一个空闲块。



现在我们需要做的是找到一个包含我们要覆盖的信息的使用中的块。 如果我们把这个块放在自由列表中最适合的块，以便进行后续的受控分配，我们将把自己的数据分配到这里，而不是使用块的数据，有效地替换了块的内容。 这意味着我们可以任意替换任何使用中的块的内容。

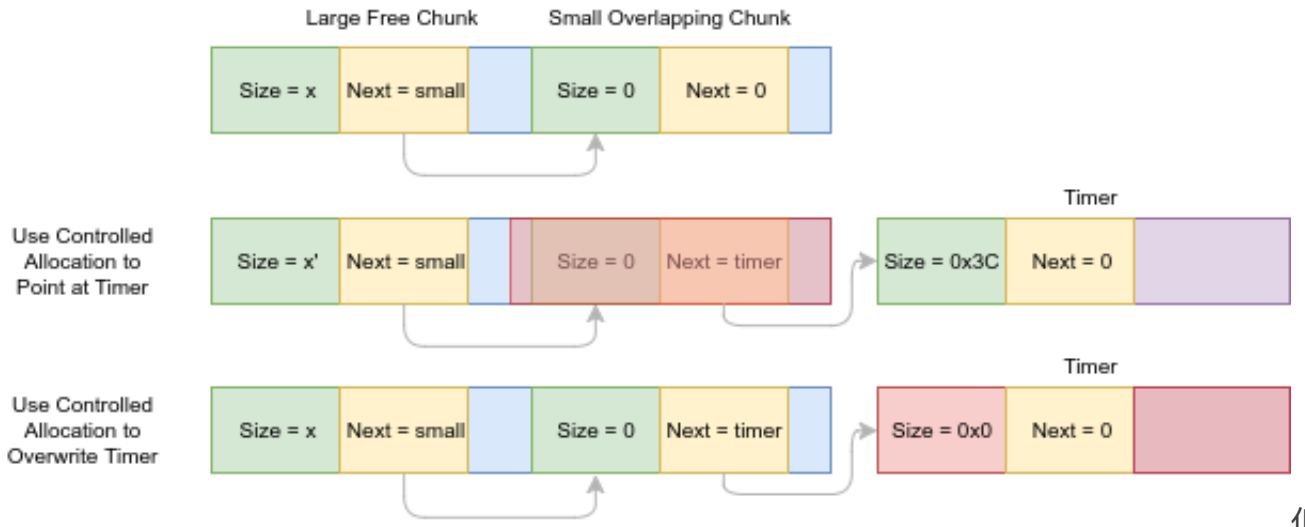
由于我们希望实现完整的代码执行，因此定位和覆盖堆中的函数指针是有利的。 但是，我们在哪里可以期望在堆上找到这样的值？ 那么对于初学者来说，Wi-Fi标准中有一些必须定期处理的事件，例如对相邻网络执行扫描。 假设固件支持使用通用API来处理这样的定期定时器，这可能是一个安全的假设。
由于定时器可能在固件操作期间创建，因此它们的数据结构（例如，执行哪些功能和何时）必须存储在堆上。为了定位这些定时器，我们可以对IRQ向量表项进行逆向工程，并搜索与处理定时器中断相对应的逻辑。 在这样做之后，我们找到一个条目的链接列表，其内容似乎与 brcms_timer 结构相符，在brcmsmac（SoftMAC）驱动程序中使用。 在写短脚本后，我们可以转储给定RAM快照的计时器列表：

```
Timer   : 0x00180708
Timeout : 0 milliseconds
Function: 0x0023ce24
Argument: 0x0023cb40
Expired : True
------------------------------
Timer   : 0x0023DA30
Timeout : 0 milliseconds
Function: 0x00000000
Argument: 0x00000000
Expired : False
------------------------------
Timer   : 0x0023C808
Timeout : 830 milliseconds
Function: 0x000081ad
Argument: 0x0023c7f8
Expired : False
------------------------------
Timer   : 0x00217F48
Timeout : 1168 milliseconds
Function: 0x000081ad
Argument: 0x00217f38
Expired : False
------------------------------
Timer   : 0x00214688
Timeout : 21302 milliseconds
Function: 0x000081ad
Argument: 0x00214678
Expired : False
```

我们可以看到定时器列表是按超时值排序的，大多数计时器的超时时间相对较短此外，所有定时器在固件的初始化期间被。分配 ，因此存储在恒定地址处。 这很重要，因为如果我们想以定时器定位我们的空闲块，我们需要知道它在内存中的确切位置。

所以剩下的就是使用我们的两个原始语句用我们自己的数据来替换上面的一个定时器的内容，将定时器的功能指向我们选择的地址。

这是游戏计划。首先，我们将使用上述技术创建两个重叠的空闲块。 现在，我们可以使用受控分配原语来将上面列出的定时器之一的较小的空闲块指向。接下来，我们创建另一个受控分配（释放旧的分配）。这个大小为0x3C，定时器块最适合。 因此，在这一点上，我们将覆盖定时器的内容。

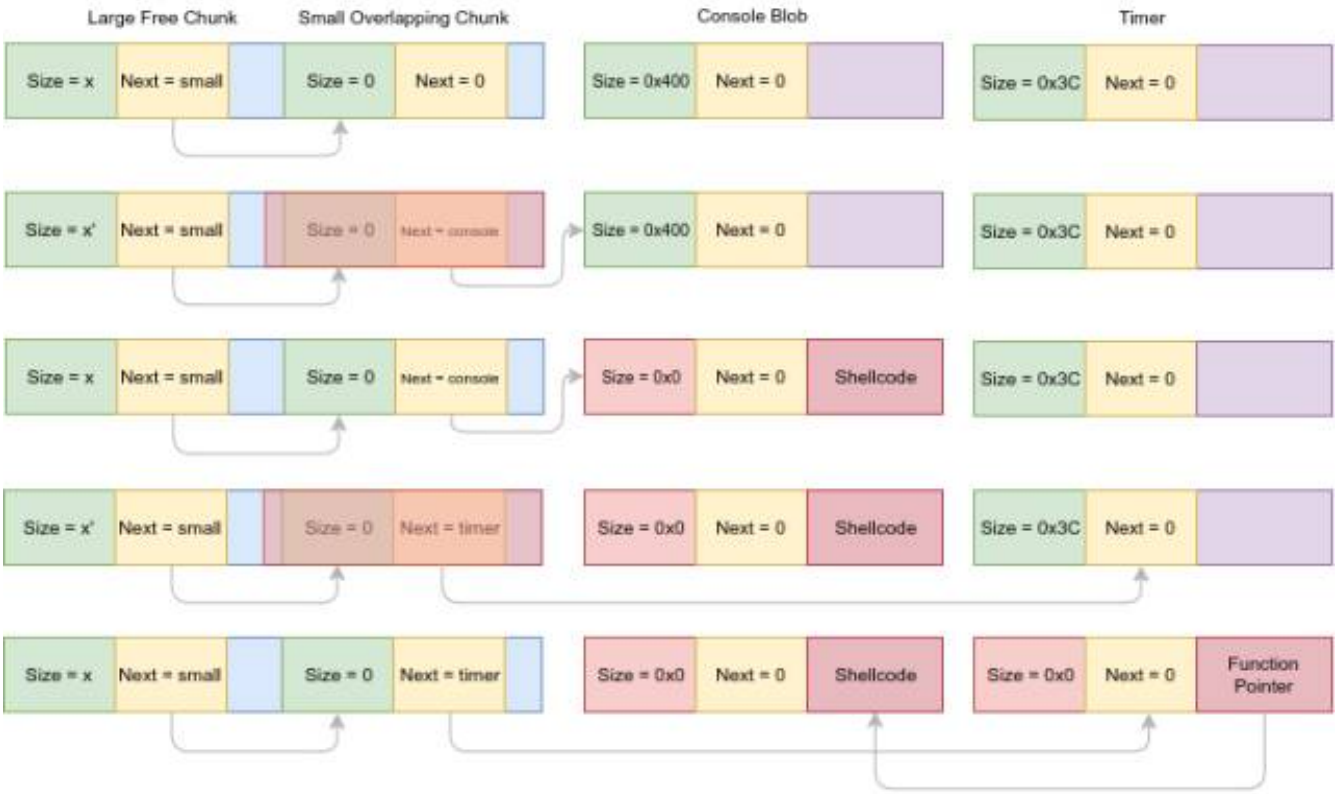


但是我们将定时器指向哪个功能？ 那么，我们可以使用同样的技巧来命令堆上的另一个使用中的块，并用我们自己的shellcode覆盖它的内容。在简单搜索堆之后，我们遇到了一个大块，它在芯片的引导顺序中只包含控制台数据，然后被分配但未被使用。 不仅分配相当大（0x400字节），而且它也被放置在一个恒定的地址（因为它在固件的初始化序列期间被分配） - 完美的我们的利用。

最后，我们如何确保堆的内容甚至可执行？ 毕竟，ARM Cortex R4有一个 Mioory P rotection Unit （MPU）。与MMU不同，它不允许虚拟地址空间的便利，但它允许控制RAM中不同内存范围的访问权限。 使用MPU，堆可以（应该）被标记为RW和不可执行。
通过颠倒二进制中固件的初始化例程，我们可以看到MPU在启动过程中确实被激活。 但它配置的内容是什么？ 我们可以通过编写一个小程序存根来找出主板的内容：
0x00000000 - 0x10000000
AP：3 - 完全访问
XN：0
0x10000000 - 0x20000000
AP：3 - 完全访问
XN：0
0x20000000 - 0x40000000
AP：3 - 完全访问
XN：0
0x40000000 - 0x80000000
AP：3 - 完全访问
XN：0
0x80000000 - 0x100000000

AP：3 - 完全访问
XN：0

当MPU被初始化时，它被有效地设置为将所有内存标记为RWX，使其无用。 这样可以节省我们的麻烦...我们可以方便地从堆中直接执行我们的代码。
所以，最后，我们有一个漏洞准备好了！ 把它们放在一起我们现在可以劫持一个代码块来存储我们的shellcode，然后劫持一个计时器来指向我们存储的shellcode。 一旦定时器到期，我们的代码将在固件上执行！



最后，我们已经经历了整个研究平台的过程，发现一个漏洞并撰写了一个完整的漏洞。 虽然这篇文章比较长，但是 我留下的细节 还有 很多 细节。 如果您有任何具体问题，请通知我。 您可以在这里找到完整的漏洞，包括 说明 。 漏洞利用包括一个相对良性的shellcode，它只是在固件的RAM中写入一个魔术值来寻址0x200000，表示成功执行。

总结

我们已经看到，Wi-Fi SoC上的固件实现非常复杂，但在安全性方面仍然落后。 具体来说，它缺乏所有基本的漏洞利用缓解 - 包括堆栈cookie，安全断开和访问权限保护（通过MPU）。
Broadcom已经通知我，较新版本的SoC使用了MPU以及几个额外的硬件安全机制。 这是一个有趣的发展，是朝正确方向迈出的一步。 他们也在考虑在未来的固件版本中实

施漏洞利用缓解。
在下一篇博文中，我们将看到如何使用我们对Wi-Fi SoC的假设控制，以便进一步升级我们对应用处理器的权限，接管主机的操作系统！