Reverse-Engineering DisplayLink devices

Florian 'floe' Echtler, Chris 'platon' Hodges

December 28, 2009

Outline

DisplayLink Hardware

Say hello to DisplayLink:

- features: pretty cheap, DVI output, magic *compression*!
- So let's look at the protocol: install driver on WinXP in VirtualBox, attach device to VM, start `usbmon`.
- Unfortunately..

Encryption

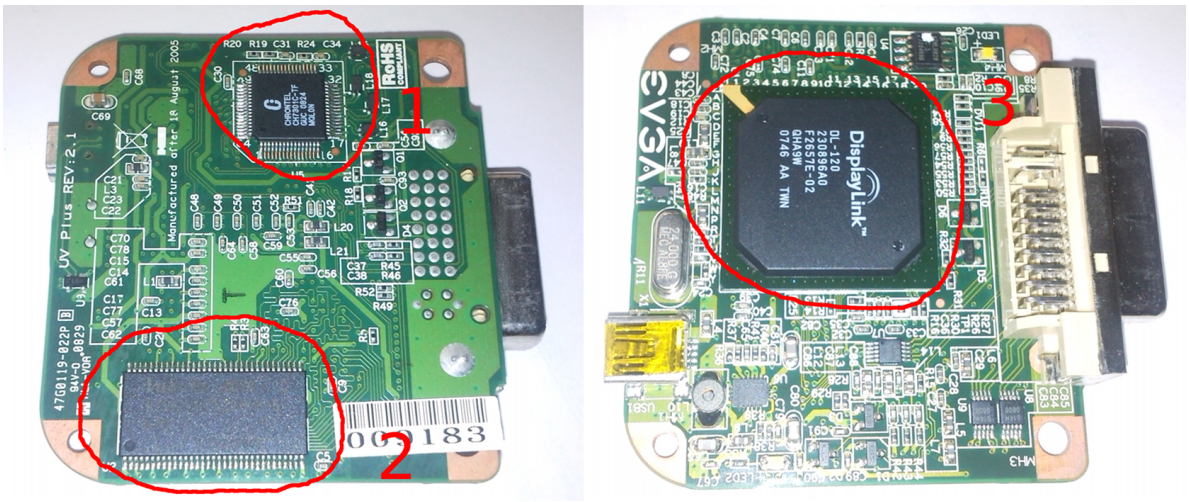
..it's all encrypted!

USB Dump: bulk transfers

```
S Bo:4:122:1 -115 8192 = eb88b508 afd71fa5 704418d1 da3c920d ee5ba235 b429d465 2f80de90 0e35c9bf
S Bo:4:122:1 -115 8192 = e56107e7 3fa5df64 397e1c1b a20d417b 8135b460 f77b80a0 fb90a1ba 86edbd27
S Bo:4:122:1 -115 2560 = 0edb3fa5 df64397e 1c129e0d 417b8135 b460fe5f 80a0fb90 a1ba86e4 99279909
S Bo:4:122:1 -115 16384 = 1ee7f414 85975c2e a39601a8 801936cb 613e0df0 14b01b04 48bdfd55 64f38e50
```

What now?

DisplayLink internals



1. one DVI encoder (Chrontel CH7301)
2. one 128 MBit SDRAM (Hynix HY5DU281622ETP)
3. and one *HUGE* ASIC (DisplayLink DL-120)

Without an electron microscope: tough luck.

Outline

More of the same, please

- first idea (no-brainer): just dump the same data to the device again
- Python script + `pyusb`.. and voilá: a Windows desktop image!
- same log works on different devices
- $\longrightarrow$ no hardware-dependent encryption

More of the same, please - Results

- several small blocks (10b - 4kB), two big blocks (150 - 300kB)
- blocks have to be sent in correct sequence
- device crashes otherwise
- $\longrightarrow$ stream cipher? $data[n]$ xor $key[n\%keylength]$
- first big block clears framebuffer to black
- second big block contains desktop image

Init Sequence: EDID

- init sequence partly unencrypted (control transfers)
- e.g. EDID readout

USB Dump: EDID readout

```
S Ci:4:122:0 s c0 02 0000 00a1 0040 64 <
C Ci:4:122:0 0 64 = 0000ffff ffffffff 0038a38e 66010101 012c0f01 0380261e 78ea1145 a45a4aa0
S Ci:4:122:0 s c0 02 3f00 00a1 0040 64 <
C Ci:4:122:0 0 64 = 00701300 782d1100 001e0000 00fd0038 4b1f510e 000a2020 20202020 000000fc
S Ci:4:122:0 s c0 02 7e00 00a1 0003 3 <
C Ci:4:122:0 0 3 = 000053
```

Init Sequence: random garbage

- 16 seemingly random bytes in init sequence
- change after each initialization
- but: repeat sometimes after VM restart
- smells like a crypto key

USB Dump: suspicious data

```
S Co:4:122:0 s 40 12 0000 0000 0010 16 = 2923be84 e16cd6ae 529049f1 f1bbe9eb
C Co:4:122:0 0 16 >
```

Random Garbage continued

Sample random sequences

- `29 23 be 84 e1 6c d6 ae 52 90 49 f1 f1 bb e9 eb`
- `f6 22 91 9d e1 8b 1f da b0 ca 99 02 b9 72 9d 49`
- `b3 12 4d c8 43 bb 8b a6 1f 03 5a 7d 09 38 25 1f`
- ...

Chris' idea: just Google for these..

Random Garbage continued

Chris' idea: just Google for these

- much to our surprise: hey presto, hits!
- how is this possible? Google results are from many different contexts..
- solution: this is simply output of the default Microsoft RNG!
- conclusion: this *is* the crypto key

Reference: `http://www.maushammer.com/systems/dakotadigital/lcd-usb.html#authentication`

Finding the Crypto Algorithm

- we have the key - now what about the algorithm?
- basic cryptanalysis: compare cryptotext with itself

Pseudocode

```
unsigned data[len*2]
int counter[n] = { 0, 0, 0, ..., 0 }

for i = 1 to n
  for p = 0 to len
    if data[p] == data[p+i]
      counter[i]++
  endfor
endfor
```

Finding the Crypto Algorithm

- try with first big block of encrypted data
- unmistakable maximum: 4095 (and multiples) $== 2^{12} - 1$
- points to a certain class of pseudo-RNG:
- *linear feedback shift register (LFSR)*

Results (for n in 1..8192)

shift 8190: count 6333
shift 4095: count 3148
shift 7631: count 49
shift 7748: count 48
...

Rebuilding the Crypto Algorithm: first steps

assumptions so far:

- basic stream cipher
- key generator: maximal 12-bit LFSR

now let's find the keystream generator!

- it's time for the disassembler (IDA Pro freeware edition)
- string analysis shows: driver uses `libusb`, statically linked
- (probably raising some interesting licensing questions)
- data submitted through `usb_bulk_write`
- try to work backwards from there

Rebuilding the Crypto Algorithm: results

- [insert looong weekend spent reading assembler and cursing at virtual functions]
- finally, try a much simpler approach:
- search for immediate value 0x0FFF == 4095!
- found: subroutine with three nested loops which generates 4095 bytes of data
- contains test against 0x0829 == 0000 1000 0010 1001 == $x^{12} + x^6 + x^4 + x^1$
- this is a generator polynom for a maximum 12-bit LFSR
- found the keystream generator!

Rebuilding the Crypto Algorithm: results

- keystream is always the same.. so where does the random 16-byte key fit in?
- LFSR generates keystream (offset $\rightarrow$ value), but also reverse-mapping table (value $\rightarrow$ offset)
- starting offset for keystream is taken from RMT
- index for RMT is created from key through CRC routine
- generator polynom 0x180F == 0001 1000 0000 1111 == $x^{12} + x^{11} + x^3 + x^2 + x + 1$ (standard CRC12)

Rebuilding the Crypto Algorithm: results

- $\longrightarrow$ CRC12 of key is starting value for LFSR
- $\longrightarrow$ key with CRC12 of zero should disable encryption!
- doesn't work with every one
- but: driver contains flag for debug mode (found by Chris)
- when enabled, one of several default keys with CRC12 $==$ 0 is used

Default Keys

- `47 3d 16 97 c6 fe 60 15 5e 88 1c a7 dc b7 6f f2`
- `57 cd dc a7 1c 88 5e 15 60 fe c6 97 16 3d 47 f2`

Outline

Command Overview

GFX Command

`0xAF 0x60` + flags + subcommand:

- `0x10`: Huffman-encoded data
- `0x08`: 16-bit mode
- `0x02`: bitblt
- `0x01`: write RLE-encoded data
- `0x00`: write raw data

followed by 3-byte target address, 1 byte pixel count & data

Command Overview

Auxiliary Commands

- set register: `0xAF 0x20 reg val`
- flush buffer: `0xAF 0xA0`
- set Huffman table: `0xAF 0xE0 ...` (see below)

References: `http//floe.butterbrot.org/displaylink/`
`http://github.com/floe/tubecable`

libdlo

nominally open-source..

- ..but *still* with encrypted init sequences..
- ..and without compression.

hence: back to the drawing board.

Compression Variants

- run-length compression: trivial, used for clearing the buffer
- Huffman-style compression: not so trivial, only replay possible so far
- Observation: 4.5 kB binary blob sent by command `0xAF 0xE0`
- Huffman decompression fails without this blob $\rightarrow$ decompression table?

Finding the Huffman tree

First Results

- analyze black/color stripe patterns sent by Windows driver
- result 1: does not encode pixel values, but difference to previous pixel value
- result 2: bit sequence 00 = difference value 0

Cryptanalysis again

- approach: treat this as another crypto problem
- find mapping: pixel value $\longrightarrow$ Huffman sequence
- chosen-plaintext attack: send suitable color pattern for every difference value
- 65536 differences are not so many: bruteforce!

Finding the Huffman tree

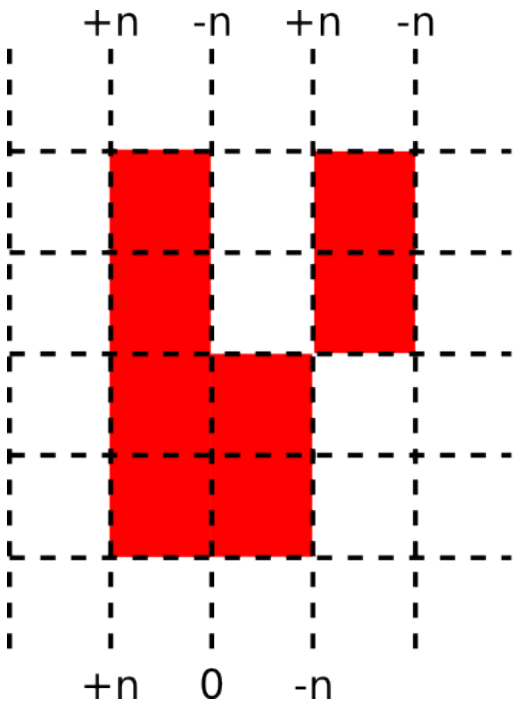


Figure: Huffman-bruteforce pattern

Finding the Huffman tree

- generate pattern for every 16-bit color n on a black background
- results in two distinct bit sequences:
- $00 n_+ n_- n_+ n_- 0000..$ and $00 n_+ 00 n_- 0000..$
- strip 2 leading zeroes
- find longest repeating substring in first sequence $\longrightarrow n_+ n_-$
- find longest common prefix with second sequence $\longrightarrow n_+ \ / \ n_-$
- sanity check: n_+ and n_- should be of same length

Finding the Huffman tree

- automate with a bit of shell scripting
- add some small string analysis tools
- retrieves codes for 32 color values in one iteration
- total running time about 4 hours
- about 50 values split across command boundaries
- repeat with slightly shifted pattern

Outline

Missing pieces

Two unsolved questions

- 8-bit Huffman sequences: pretty trivial, just tedious
- compressed Huffman table: the last big riddle

How can a binary tree with 30 levels and 2^{16} leaf values be represented in 4.5 kB?

Compressed Huffman table

Structure seems to consist of 512 items with 9 bytes each.
Sample entries:

Compressed Huffman entries

- Last 135 entries:
  `0xff, 0xff, 0xff, 0x7f, 0xff, 0xff, 0xff, 0xff, 0x7f`
- Entry possibly corresponding to leaf nodes 2113 and 2081 (leaf depth 6):
  `0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x20, 0x00, 0x06`
  `0x08, 0x41, 0x01, 0x00, 0x00, 0x08, 0x21, 0x01, 0x00`
  `0x08, 0x61, 0x00, 0x06, 0x12, 0x00, 0x00, 0x00, 0x06`

The End

Thanks to..

- Roberto de Ioris, Marcus Glocker, Sven Killig & others: for being brave enough to use this stuff :-)
- DisplayLink: for donating devices and grudging acceptance
- Chris Hodges: for being a cool co-hacker
- you: for your attention!

Questions & comments are welcome - contact us at `floe ÄT butterbrot.org` and `chrisly ÄT platon42.de`.

That's all folks!