# Reverse-Engineering DisplayLink devices
USB to DVI for Hackers

Florian Echtler <floe@butterbrot.org>
Chris Hodges <chrisly@platon42.de>

November 27, 2009

**Abstract**

DisplayLink produces nice, useful USB graphics adapters. Unfortunately, they had no real Linux support. In this paper, we'll describe how we first reverse-engineered the encryption and basic protocol, prompting DisplayLink to actually release a Linux driver on their own. However, this driver still didn't support compression. In the second part, we'll therefore describe how we reverse-engineered the compression algorithm.

## 1 Introduction

Say hello to Displaylink:

Their devices have some nice features: pretty cheap, DVI output, magic *compression*! So let's look at the protocol: just install the driver on WinXP in VirtualBox, attach the device to VM and start `usbmon` on the host. Unfortunately, when you look at the following example of some captured bulk transfers for a black screen, it quickly becomes apparent that..

```
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
```

..it's all encrypted!

What now?

First step, of course, is to have a look at the internals of the device (see figure 1):

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.

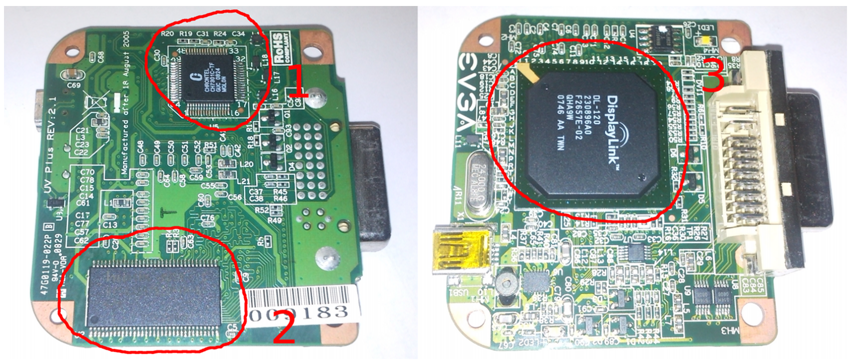

Figure 1: Displaylink internals

## 2 Cracking the Encryption

### 2.1 Replay Attack

The first and most simple approach to getting rid of the encryption is really a no-brainer: try a replay attack and just dump the same data to the device again. We hacked together a Python script to parse the `usbmon` dump and send it through `pyusb`, and presto: a Windows desktop image! We verified that the same log works on different devices (DL-120 and DL-160), consequently the encryption doesn't seem to contain any device-specific components.

A "standard" init sequence (from device connection to static desktop image) seems to consist of several small blocks (10b - 4kB) and two big blocks (150 - 300kB). These blocks have to be sent in the original order, otherwise, the device crashes. This indicates a stream cipher ($data[n]$ xor $key[n\%keylength]$) which gets out of sync and causes some internal checks to fail if the blocks are sent in a different order. However, it was possible to insert arbitrary delays between the blocks, which showed that the first big block clears the framebuffer to black and the second big block contains the desktop image

### 2.2 Finding the Crypto Key

As several repeated initializations yield different sequences, it's obvious that a crypto key has to be hidden somewhere in the init sequence. The very first part of the initialization consists of control transfers which appear to be unencrypted. One example is the following readout of the monitor info (EDID).

```
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
```

Even more interesting are 16 seemingly random bytes in the first part of the init sequence which change after each initialization. However, as we had the driver running inside the VM, we noticed that these bytes repeat sometimes after a VM restart. Safe to say that this smells like a crypto key.

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

Some more samples of these 16 random bytes:

- `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`
- ...

At this point, Chris had a great idea: just Google for these hex strings. Much to our surprise, Google actually produced several hits! How is this possible? Even more confusingly, the results are from many different contexts.. The solution, as explained by this website:[1] this is simply output of the default Microsoft random number generator! Conclusion: this *is* the crypto key.

## 2.3 Finding The Crypto Algorithm

So we found the key - now what about the algorithm? To get some ideas, we did a very basic type of cryptanalysis and just compared the 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
```

We did this with the first big block of encrypted data (which clears the framebuffer to black) and arrived at the following results (for n in 1..8192):

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

Obviously, there's an unmistakable maximum at 4095 (and multiples). This allows the conclusion that the key period is 4095, which is $2^{12}-1$. A period of $2^n-1$ is a strong indicator for a certain class of pseudo-random number generators which are called *linear feedback shift register (LFSR).*

## 2.4 Reconstructing The Crypto Algorithm

At this point, we have two basic assumptions: a) we are dealing with a basic stream cipher, which is b) generated through a maximal 12-bit LFSR. What is needed now is to actually find the keystream generator. Therefore: it's time for the disassembler (IDA Pro freeware edition). A quick string analysis showed that the driver actually uses `libusb`[2]. In `libusb`, bulk data is submitted through `usb_bulk_write` - our first approach was to try to work backwards from this call.

[insert looong weekend spent reading assembler and cursing at virtual functions]

After this approach didn't provide really convincing results, we tried a much simpler approach, which was to search the assembler code for the immediate value `0x0FFF` = 4095 (as the key routine obviously will generate this much data). Consequently, we found a subroutine with three nested loops which generates 4095 bytes of data. Within this subroutine, we also located a test against `0x0829` = 0000 1000 0010 1001 = $x^{12} + x^6 + x^4 + x^1$. Some textbook reading reveals that this is a generator polynom for a maximum 12-bit LFSR. So we've got the keystream generator!

One interesting observation is that the keystream is always the same.. so where does the random 16-byte key fit in? A closer look at the LFSR routine revealed that it does not only generate the keystream (offset → value), but also a reverse-mapping table (value → offset).

[1] `http://www.maushammer.com/systems/dakotadigital/lcd-usb.html#authentication`
[2] which might generate interesting licensing issues

The start offset for the keystream is taken from this reverse-mapping table, and the index into the RMT is in turn generated from the 16-byte key through a CRC routine. The CRC generator polynom is `0x180F` = 0001 1000 0000 1111 = $x^{12} + x^{11} + x^3 + x^2 + x + 1$ (standard CRC12).

As the CRC12 of the 16-byte key is therefore the starting value for the LFSR, a key with a CRC12 of zero should disable encryption! Our first tests with some generated keys didn't work, however, Chris identified a debug mode flag in the driver. When enabled, one of several default keys with CRC12 = 0 is used (keys were taken from different driver versions):

- `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`

## 3 The Graphics Protocol

Now that we had disabled the encryption, it was pretty easy to figure out the basics of the communications protocol by simply setting various desktop backgrounds in the Windows XP VM and observing the resulting USB dumps. We won't go into much detail regarding the command set here, for a complete reference, please see `http://floe.butterbrot.org/displaylink/` and `http://github.com/floe/tubecable`. One important detail, however, is that the device manages two separate framebuffers in its memory space, one for 16-bit color depth (RGB565) and an additional 8-bit framebuffer (RGB323) for combined 24-bit color depth. Every graphics command (raw write, RLE write, bitblt..) comes in an 8-bit and a 16-bit version for each of the two framebuffers.

One thing still missing at this point, however, was the Huffman-based graphics compression.

### 3.1 Intermission: DisplayLink's Reaction

After we had published this on the web, nothing happened for a while. Then, quite soon after some people had started actually implementing drivers, Displaylink announced their own opensource library, `libdlo`. While this is generally laudable, the first version was *still* with encrypted init sequences and without compression.

Hence: back to the drawing board.

### 3.2 Huffman Compression

There is a run-length compression mode which is pretty trivial (it's actually used by the Windows driver to clear the framebuffer at initialization). However, the Huffman-style compression is not so trivial. While we had logged many compressed images and could also replay them, we had so far not been able to compress our own data, as the Huffman table was still missing.

During the secondary init sequence, we observed a 4.5 kB block of data with no apparent purpose (in fact, this block even is preceded by its own command `0xAF 0xE0` which is used nowhere else). When this block is omitted, everything works as before - with the exception of Huffman-based compression. Obviously, this block contains the Huffman table. Unfortunately, this table itself is in compressed form, and a search in the driver didn't reveal any clues to the memory location of the uncompressed version.

However, the goal of finding the Huffman table can actually be interpreted as another crypto problem. Huffman-encoded data is a kind of ciphertext, whereas the raw, uncompressed data is the plaintext. And as the uncompressed data is just the screen contents of the VM, we can perform a chosen-plaintext attack!

From looking at the encoded results of various color stripe patterns, it quickly became apparent that the encoding scheme isn't compressing pixel values, but pixel value differences, e.g. in the range of -32767 to 32768 for 16-bit data. Another rather obvious result was that the 2-bit sequence `00` encodes the value 0. Unfortunately, it wasn't so obvious how any of the other sequences looked. A bit of knowledge about Huffman codes at least allows to conclude that every other pattern either has to start with `1` or `01`.

On the other hand, that's just 65535 pairs of value → bit-sequence. The simple solution: brute force!

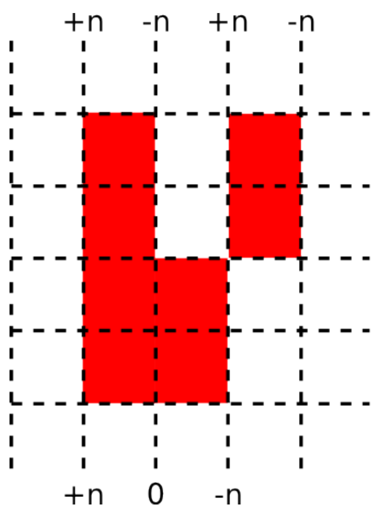

Figure 2: Huffman-bruteforce pattern

The pattern in figure 2 is generated for every 16-bit color n on a black background. This results in two distinct bit sequences: the first one is $00n_+n_-n_+n_-0000..$, the second one is $00n_+00n_-0000...$ By finding the longest repeating substring in the first sequence, the pattern $n_+n_-$ can be isolated. By finding the longest common prefix with the second sequence, the pattern $n_+$ can be retrieved and consequently also the pattern $n_-$.

With a bit of shell scripting and some small string analysis tools, this process was automated to retrieve the codes for 32 color values in one iteration. Total running time was about 4 hours; about 50 values had to be retried with a slightly modified image, as the bit patterns had been split across command boundaries.

## 4 Future Work

There's two things missing from the picture. First, the Huffman table has only been created for 16-bit mode - the above process should also be done for 8-bit mode where a single 0 bit encodes the difference value 0. The second thing is the encoding scheme which is used to send the Huffman table to the device - in memory, the table occupies about 320 kB, but is transmitted as a blob of only 4.5 kB.