Fri, 07 Nov 2008

Anatomy of an Optical Medium Authentication (Part 1)

By tmbinc

Filed under Hacking Projects

Introduction

Abstract

In this series of articles, I will talk about the design, implementation and fall of an optical media authentication used on a popular, but past, gaming console. I will show that it's possible to reverse engineer such stuff without access to expensive equipment or insider information.While I will not talk about practical implementation of attacks against the discussed scheme, I will show that this has been done, and I will analyze how this has been done. More after the break.

Disclaimer

(First, a disclaimer: I don't intend to break the law. This won't be a "how to break a copy protection". Quite different, this will be a "how a copy protection works". You think that's the same? It isn't; or at least, it shouldn't. An optical media copy protection is usually not based on obfuscation, contrary to popular belief; this particular protection scheme has been documented in various patents ([1], [2]), so I assume it's safe to guess that a knowledge of the technology doesn't allow you to create your own "authentic" discs. To take it back to a technical level, the hearth of the technology is an asymmetric cipher in combination with a property which you can only generate randomly, but not intentional. Finally: If you are here because you want to break copyright law: a.) go away! and b.) there are much easier ways thanks to the constant screw-ups of the firmware people.)

Introduction

I'm interested in optical storage technology since a long time ([3]), and one of the better understood systems is the Gamecube optical drive. Quite unique is the media-based copy protection, which, as far as I know, isn't used in any other system. The goal of this copy protection is, like most other optical media copy protections, to restrict the drive to only read discs which were manufactured in an "authentic" way.

Let's understand DVDs…

To understand the way this copy protection works, we first need to revisit how DVDs work at all. First, open up the DVD specs from ECMA - they are free, and quite easy to understand (if you ignore the heavier parts). That will it make easier for you to understand. References to figures or sections are for this document.

…from the filesystem…

Let's start from the top: the user visible image. It's usually formatted in some filesystem, like ISO9660, but that doesn't really matter. All the DVD format cares about is that it's a number sectors, where each sector is 2048 bytes long. And, for reasons we will see later, the sector count must be a multiple of 16. Each sector is packed into a "Data Frame". If you follow the ECMA-Specs, this is described in Section 4, 16.

Each data frame has a size of 2064 bytes; 16 bytes more than the payload. The first 4 bytes of each data frame is called ID, and contains, next to some flags, the Physical Sector Number (PSN). Each data frame, and, as we see later, all other frames, have an associated, hopefully unique, PSN. Several ranges of PSNs are reserved for special data (more about that… yes, later), but let's just say that the data frame containing the first user sector has the PSN 0x30000, the second one 0x30001 etc. You get the idea.

When the drive searches for a specific sector, it will decode the ID values of the incoming datastream until it reaches the requested PSN (additional seek algorithms might move the pickup if the PSN is too far away, or already passed). Because the drive needs to decode the correct ID on the fly, it must be protected with an error-correction code. Thus, the next 2 bytes, called IED (ID Error Detection Code), store a rs(6,4)-code (Reed-Solomon) of the ID field. This helps the drive to correct read errors of the ID field.The next 6+2048 bytes are payload. The final 4 remaining bytes is an error detection code, called EDC. It cannot be used to recover broken data, but it serves as a last way to detect uncorrected data. It's calculated over the rest of the data, including ID, IED, the magic 6 bytes and the 2048 bytes of payload data.

The 2048 bytes, starting at offset 12, are the user payload. The magic 6 bytes are not really documented. They are used in DVD-Video applications, but we don't need to care

about them. They are just there.The 2048 main bytes have an additional property; they are scrambled with an LFSR-based bitstream, to prevent DC. The LFSR-sequence is documented in section 17 of the ECMA-docs, and have one parameter, the “initial pre-set value”, or “seed”. This value is based on some bits of the PSN, so the scrambling pattern isn’t completely static.

… over Error correction codes …

The next step puts those “Data Frames” into so-called “ECC Blocks”. This is the magic of the DVD error recovery, and uses two reed-solomon code, PI and PO, which is added to the data frames. PI works on rows, PO works on colums. 16 data frames together are packed to form an ECC block. Because of this interleaving, burst errors, i.e. defects on a number of consecutive bits, are spread over the ECC block. This is important, because each error-correction code can only fix a limited number of bits.

…to the raw bits.

Finally, those ECC blocks are re-arranged into so called “recording frames”. Finally, sync-words are added between these recording frames (32bit long words), and the result will be EFM+-encoded. EFM+ stands for “Eight-to-fourteen-modulation-plus”, and is a revised version of the code used on the Compact Disc. On a compact disc, 8 payload-bits would be encoded into 14 bits, with special properties for easier decoding, like a minimum and maximum distance between ones. Additionally, 3 zero bits need to be inserted between two encoded 8bit-words. EFM+ differs from that that these gap bits are no longer required, but instead the generated code is 16 bits long. Thus it’s more a “eight-to-sixteen-modulation”. The important part here is that the phyiscal encoding is twice as large as the payload size. We will need this information later.

The EFM+-decoded data is then NRZI-encoded and written to disc; here, pits and lands are created.

Leadin/Leadout

Additionally to the user data zone, there is a leadin and a leadout. The lead-in zone, i.e. all PSNs < 0x30000, contains a number of differently formatted frames. Most important are the PFI- and DMI-frames, which both carry meta-information about the data zone (for example the size).

Gamecube differences

The scrambling "hack"

Gamecube optical discs (short: GOD) are "nearly" DVD-compliant. They have a normal leadin (PSN < 0x30000), but the data section uses a slightly different scrambling seed algorithm. Remember the scrambling done to the 2048 data bytes, where different sectors have a different scrambling pattern. On a normal DVD, you would use the the "initial pre-set number", which is the second-least significant nibble of the sector number (ID7..4) to index into a seed table to give you the "initial pre-set value", which is the start value of the LFSR. This can be simplified if you see the "scrambling stream" as a 32767 bytes string, generated by the LFSR with the start value (1), just that you don't stop after generating 2048 bytes. Instead, you continue with the LFSR, until it repeats (which is after 32767 iteration, where each iteration gives you one byte). You would then offset into this table with the ID7..4*0x800. That means that sector 0x30000 would use bytes 0..2047 of this table, sector 0x300010 would use 2048..4097 etc.

Now GODs use the same scrambling string, however, with an additional offset of 0x3C00. Thus, 0x30000 is scrambled with bytes 0x3c00..0x43ff. Furthermore, ID7..4 must be XORed with a per-game constant when reading PSN >= 0x30010. The game-specific constant is based on a simple checksum of the first 6 bytes of the user-image ("gameid") - the first ECC block (PSN 0x30000) uses zero instead. Thus, the offset for a sector into the scrambling table isn't ID7..4*0x800 anymore, but 0x3c00 + (ID7..4 ^ gameid-checksum)* 0x800. (Yes, this might be > 32767. Just repeat the string in these case)For PSNs less than 0x30000, i.e. the lead in, the original DVD scrambling seed is used. (As a side note, "NR-Media", i.e. DVD-Rs specially burned for development, don't have the 0x3C00-offset, but instead use a XOR value of 0x9 for the whole media, including the lead-in. That makes it so much more complicated to dump those discs using a PC drive, because it cannot even read the PFI/DMI.)

But all of this is just obfuscation. It makes it harder, but still possible, to read the actual content from the disc using a PC DVD-ROM.

Identification data

The lead-in of a GOD, for example the DMI (PSN 0x2f801, for example, but these information are repeated over the whole control zone, which is a just one part of the lead-in), can be read using a normal DVD-reader, without special tools. If you see the string "Nintendo Game Disk", then you hit the right sector. The DMI isn't specified in the ECMA format at all.

Working around by modifying the reader

But there is "one more thing": A GOD has a different Data Frame layout. Instead of not

using the magic 6 bytes, they shifted the whole user data 6 bytes to the front. That means that there is no scrambling applied to the first 6 bytes of each sector. Each user sector is still 2048 bytes; it's just that the last 6 bytes (before the EDC) are unused, not those in front of the user data.In end of 2004, a modchip called "Viper" was introduced, which made it possible to modify a gamecube to read standard DVDs. This was accomplished by basically applying 3 different patches to the drive firmware:

1. Moving the user data from offset 6 (inside the data frame) to offset 12,
2. Fixing the scrambling seed to be DVD-compliant again, and
3. removal/skip of the copy protection.

The actual copy protection

The third part is completely unrelated to the first two; contrary to popular belief, the copy protection is not based on making the disc incompatible with standard DVDs; this alone would help against consumer DVD burners, but not against profesionally manufactured copies. When mastering DVDs, it's no problem to master custom data frames. An additional feature of GODs is the usage of the "burst cutting area", often incorrectly described as "barcode". If you look into Annex H of the ECMA-specs, you'll notice that this is in fact an optional, but standard, feature. Many PC-DVD-Readers (especially burners) can read BCAs. A BCA can store up to 188 bytes of data. The BCA of my copy of the widely popular game "Phantasy Star Online" looks like the following:

```
0xA9, 0x20, 0x98, 0x65, 0x92, 0xF5, 0x12, 0x2C, 0xB6, 0xBE, 0x05, 0x37,
0x1C, 0x0C, 0x08, 0x5D,0x3B, 0x48, 0x69, 0x6E, 0x67, 0xA4, 0xB5, 0x6A,
0xE8, 0x14, 0xE2, 0x78, 0x3E, 0xFF, 0x15, 0x17,0x40, 0x46, 0x31, 0xE6,
0xF0, 0x8F, 0x42, 0x43, 0xE4, 0x87, 0xCD, 0xAB, 0x9B, 0x3E, 0x9C,
0x26,0xC5, 0x8E, 0x38, 0x04, 0xBF, 0x6B, 0x3D, 0xD7, 0x37, 0xFB, 0xFE,
0xC0, 0x33, 0x05, 0xC5, 0xC0,0x0E, 0x43, 0x6E, 0x82, 0x12, 0xB2, 0x3A,
0xF3, 0x76, 0xFD, 0x1A, 0xC7, 0x2B, 0xCD, 0x78, 0x87,0xAF, 0x4B, 0xD6,
0xA5, 0xC3, 0xFF, 0x2B, 0x7F, 0x05, 0x93, 0x2C, 0xAA, 0xD5, 0x82, 0x17,
0xB6,0x89, 0xD9, 0xE7, 0x52, 0xAC, 0x2E, 0x38, 0xA9, 0x44, 0x24, 0xE9,
0x2B, 0xA9, 0x4D, 0x23, 0xFA,0xEE, 0x07, 0x03, 0xF3, 0xED, 0xDC, 0xC6,
0x00, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,0x4B, 0x47, 0x53,
0x43, 0x01, 0x04, 0x95, 0x17, 0x21, 0x00, 0x11, 0x7D
```

BCA decryption

This doesn't look particulary helpful yet. If we look at the Gamecube drive's firmware, for example by reading the drive's memory after disc authentication, we see that the BCA is actually encrypted. After decryption by the drive, it looks like

```
|   decryption key    | |       data0         | a9 20 98 65 92 f5 12 2c b6
be 05 37 1c 0c 08 5d |              data 1 | |  decrypted data .. .. 3b 48
69 6e 67 a4 b5 6a 19 d4 03 2c 1c 02 45 aa 23 17 03 2c 03 ff 3c 30 49 43 03
2c 25 ff 3b 55 0e 26 03 2c 0d ff 3c 54 2e d7 03 2c 12 ff 4c 8c 25 c0 03 2c
17 ff 46 da 15 f3 27 ed c2 ff 3b 63 ed 87 bb 89 58 ff aa 10 db 7f 0a 48 89
ff  9e  5b  d1  b6  f4  5e  6f  ff  a3  15  d6  a9  d1  84  f5  ff  f9  b8  ..
..                  | |  BCA_78 | | user data 86 fa eb 78 5e ff 15 de ff ff
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 .. .. | 4b 47 53 43 01 04 95 17 21 00 11 7d
```

The challenges...

Much better. Most important, we see a table in the middle, containing 6 valid entries:

```
19d4 032c1c 02 45 aa 2317 032c03 ff 3c 30 4943 032c25 ff 3b 55 0e26 032c0d
ff 3c 54 2ed7 032c12 ff 4c 8c 25c0 032c17 ff 46 da
```

...and their responses:

This table is a list of PSNs, associated with a 16-bit value to each. Something with those PSNs must be special.

Using my previous hack, I could capture the bitstream of these PSNs. I modified my bitstream decoder to dump out the raw recording frames of the PSNs in question. If we, for example, look at PSN 0x32c17, we notice the following:

```
940 22 24 41 04 88 09 09 10 49 09 09 24 11 01 02 12 950 22 40 40 92
10 92 42 04 89 11 11 21 02 48 92 21 960 02 20 92 20 40 08 91 10 42 12
10 28 40 00 00 00 970 00 00 00 00 00 29 02 00 10 11 11 08 42 44 08 24
980 48 48 22 14 24 14 28 51 45 12 00 a2 08 50 84 24
```

The obviously interesting thing here are the string of zeros, which begins approx. at offset 0x96D. It consists of at least 68 bits of zeros, which is an otherwise forbidden value - the used EFM+ encoding makes sure that there is a maximum of 11 subsequent zeros are allowed between ones. This is done to keep the receiver clock in sync with the bitstream. This long string of zeros is definitely a violation of the encoding specifications.

If we calculate the relative position inside the sector, it starts at bit 0x96D*8, which is 19304. Measured in payload bits, i.e. after EFM+, this would be half of it, which is 9652. If we compare this to the value in the BCA challenge (0x25c0=9664), we notice that's it's very close to this. Random conincidence? Let's take a look at the other sectors. The left side is the result of searching zeros strings within the recording frames, using a simple tool. The right side is the information encoded in the BCA, and the diff between the found value and the encoded value:

```
PSN 00032c03, 30 zeros @231e , BCA: 0x2317 (diff=-7) PSN 00032c08, 40 zeros @18e0 PSN 00032c0d, 3c zeros @0e2a , BCA: 0x0e26 (diff=-4) PSN 00032c12, 4b zeros @2ed1 , BCA: 0x2ed7 (diff=6) PSN 00032c17, 44 zeros @25c2 , BCA: 0x25c0 (diff=-2) PSN 00032c1c, 45 zeros @19d4 , BCA: 0x19d4 (diff=0) PSN 00032c21, 3f zeros @07ec PSN 00032c25, 30 zeros @494a , BCA: 0x4943 (diff=-7) PSN 00032c2a, 31 zeros @3e96 PSN 00032c30, 40 zeros @13b2 PSN 00032c35, 46 zeros @0a9c [...]
```

Isn't this beatiful? In case you wonder, an additional, fixed offset of 0x1E has been applied, which probably accounts for the sync length.

Marks

But we haven't yet discussed how those zeros are actually introduced in the bitstream. The important part is that they are inserted *after* the DVD has been mastered. This is done in the DVD factory by cutting "marks" with a similar method used to burn the BCA into the data area. Those "marks" are roughly the same length as the BCA. There are seven of these marks in total. If you hold a gamecube disc against light, you will notice these marks. Based on our observation of the length of zeros, we can actually calculate the width of the mark: 68 zeros, times a nominal channel bitlength of 133.3 nm (see section 10.6.4), those marks have a length of approx. 9000nm, or 9 micron. The laser marks used for the BCA are "roughly 10 microns wide" [4]. It's safe to assume that the same technology is used to burn these marks as to burn the BCA.

Such a mark is also much longer than the width of the track. Thus, not only one sector is affected, but a number of sectors which are all at the same angular position. It is unclear if the drive is able to detect the angular position of a certain bit, but this would be an additional (and important!) anti-copy measure: It's nearly impossible to predict which sector bit positions align next to each other.

Actually, we can see in the log above, a much larger number of sectors is affected. To verify our theory that all of these affected bit positions are in fact from the same six marks, we can calculate the distance between zeros from two sectors. Actually we will assume that six consecutive zero-strings starting at n are from the six marks, and the n+6th string is next to the first one, just a revolution later:

Based on this assumption, let's assign a bit position of pos_mark_in_bits=PSN*19344+offset_in_payload_bits to each zero-string; 19344 is the length of a sector after EFM+-decode. The begin of the n'th-zero-string should lie right next to the beginof the n+6'th zero string. If we calculate these distances, they should correspond to the circumference of the track at the specified position. We can also assume that it will slightly increase with increasing PSNs because they are more outer, thus the circumference increases slightly.

Let's take a look at the output - the last value is the difference between this zero-string and six zero-strings ago, divided by the sector length:

```
PSN 00032c70, 40 zeros @201a <29.642525> PSN 00032c75, 3a zeros @143a <29.642680> PSN 00032c7a, 3b zeros @0256 <29.642887> PSN 00032c7e, 3b zeros @43ba <29.643455> PSN 00032c83, 38 zeros @390e <29.643300> PSN 00032c89, 4c zeros @0e36 <29.643507> PSN 00032c8e, 4c zeros @0529 <29.643455> PSN 00032c92, 40 zeros @44da <29.643507> PSN 00032c97, 3e zeros @32f9 <29.643662> PSN 00032c9c, 30 zeros @28d2 <29.643921> PSN 00032ca1, 46 zeros @1e25 <29.643869> PSN 00032ca6, 40 zeros @3ee2 <29.644127> PSN 00032cab, 48 zeros @35da <29.644386> PSN 00032cb0, 3b zeros @2a01 <29.644696> PSN 00032cb5, 30 zeros @1826 <29.645006> PSN 00032cba, 33 zeros @0dfd <29.644903> PSN 00032cbf, 3d zeros @0350 <29.644903> PSN 00032cc4, 4d zeros @240e <29.644955> PSN 00032cc9, 4d zeros @1b08 <29.645058> PSN 00032cce, 46 zeros @0f31 <29.645161> PSN 00032cd2, 3e zeros @48e9 <29.645316> PSN 00032cd7, 35 zeros @3ec4 <29.645523> PSN 00032cdc, 3e zeros @3420 <29.645988> PSN 00032ce2, 4e zeros @0952 <29.646195> PSN 00032ce7, 4e zeros @0051 <29.646454> PSN 00032ceb, 3c zeros @4009 <29.646402> PSN 00032cf0, 3b zeros @2e32 <29.646454> PSN 00032cf5, 39 zeros @240e <29.646505> PSN 00032cfa, 3c zeros @196e <29.646712> PSN 00032cff, 53 zeros @3a2a <29.646402>
```

What we see is the a measurement of the circumference; let's calculate the radius of the track at the PSN position. 29.64 sectors x (2 x 19344) bits/sector are approx. 1146712 bits per revolution. The nominal width of one bit is 133.3nm again, so we are at a circumference of 152894972 nm, or 152mm. This corresponds to a radius of 24.33mm; this is right next to the BCA, which ends at 22.5mm (see J.4.3). This also aligns with the additional marks you can see when holding the disc against light.

Upcoming: Part 2, Implementation

The drive authenticates the disc by measuring these properties, and comparing them with the values stored in the BCA. I will describe the exact details of this in Part 2 of this

series.

Upcoming: Part 3, Analysis of a successful attack

It's interesting to notice what Datel is actually doing, though this is going to be bit of speculation. They don't have visible marks, my guess is that they are embedded into the pit/land pattern. While there is an analog difference between a land and a laser-cutted mark (a difference which my setup is unable to pick up), it seems that the drive doesn't notice it either. There are more interesting properties of Datel-discs, which I will talk about in part 3 of this series.

Previous post
Next post

ps: are you still working on this?

Reply • Share ›

Shower Screen • 5 years ago

both LG and LiteOn makes a great performing dvd burner, they also feature those anti-shock mount *~;

Reply • Share ›

shuffle2 • 5 years ago

perhaps you could remake the dvd.png so it's viewable on your new(ish) theme :)

Reply • Share ›

tmbinc • 6 years ago

I totally agree. As a matter of fact, I *am* working on it, still.

Let me conclude that the second part will be much more boring that you might think. And for the third part, I'm still missing some equipment.

dasda: Could you provide an (optical :) image of this disc?

Reply • Share ›

i2c • 6 years ago

I agree with Busing that I'd like to see this article finished.

Reply • Share ›

bushing • 6 years ago

Please finish this.

Love,
Bushing

Reply • Share ›

Count Zero • 6 years ago

Excellent article indeed!

Reply • Share ›

dasda • 7 years ago

Very interesting, thanks!
I have a old Action Replay demo-disc (got it from a magazine) from Datel, it has some strange holes on the disc where those six marks are located in your image.

Reply • Share ›

Eric Ball • 7 years ago

When the Gamecube was first introduced, there was some speculation that the drive spun the discs backwards from a standard CD/DVD player. This was quickly disproven, but I wonder whether reverse spinning would have made piracy (i.e. copying via consumer DVD-Rs) more difficult. (Although it probably wouldn't have prevented counterfeitting - but almost anything can be duplicated given enough time, money, and

effort.)

Reply • Share ›

so_what · 7 years ago

hi,

i've just seen the ccc stream where you talk about the xbox360 hypervisor exploit and now im wondering how you guys managed to get the hypervisor code itself and not encrypted

Reply • Share ›

dasda · 7 years ago

Looking forward to part 2! :)

Reply • Share ›

Nate · 7 years ago

Very nice article. The best C64 floppy protection schemes used a combination of

recently

14.09.10	OpenVizsla OV3 - Hello, World!
14.08.28	OpenVizsla OV3 - FPGA design
14.05.13	OpenVizsla OV3 - Hardware
13.03.07	What's Inside: Tektronix DPO5034
13.03.07	Real Life Statistics
12.02.03	xvcd - The Xilinx Virtual Cable Dae...
11.12.22	What's Inside: Hilti PD-30
11.11.01	Almost Secure
11.10.12	What's Inside: Metz 50 AF-1 N

11.06.18 "if you call that hacking, then we e...