Hacking Sony PlayStation Blu-ray Drives

Boris Larin @oct0xor

Who am I

Security researcher at cybersecurity company

- Reverse engineering
- Finding zero-days exploited in the wild
  - CVE-2018-8174, CVE-2018-8453, CVE-2018-8589, CVE-2018-8611, CVE-2019-0797, CVE-2019-0859
- Finding supply chain attacks
  - ASUS “Operation ShadowHammer” and few others

Previously presented at:

- Rootcon, Virus Bulletin, CanSecWest, SAS, BlueHat, TyphoonCon, ISC by Qihoo 360, AVAR, Code Blue ...

Some people might also know me as @oct0xor

- Active in ps3dev scene since 2011
- Back in the days was mostly known for work on freeing DRM protected PS3 custom firmware's, developing PS3 debugging tools, and etc
- Since then also was looking at all other video game consoles but without much publicity

What this talk is about

In this presentation I will talk about security and internals of Blu-ray disc drives for Sony PS3 & PS4

Games are distributed at optical media, that's why drive should contain the best security possible

But that's also makes it a very interesting subject for security research

1. I will discuss the process of obtaining and reverse engineering the firmware
2. I will provide in-depth analysis of vulnerabilities and their exploitation to achieve code execution on multiple models of Sony PlayStation Blu-ray disc drives
3. I will talk about security features of Sony PlayStation Blu-ray disc drives

Disclaimer

This research is done purely out of curiosity and presented for educational purposes

This research does not help/support/enable/endorse to break the copyright law

TLDR:

I will be talking about security vulnerabilities, but they do not lead to full compromise of security

As far as I am aware its not possible to use my findings to circumvent copy protection

That's the reason why I am even talking about that

Blu-ray Disc (BD) - Intro

High capacity digital optical disc data storage format developed to succeed DVD

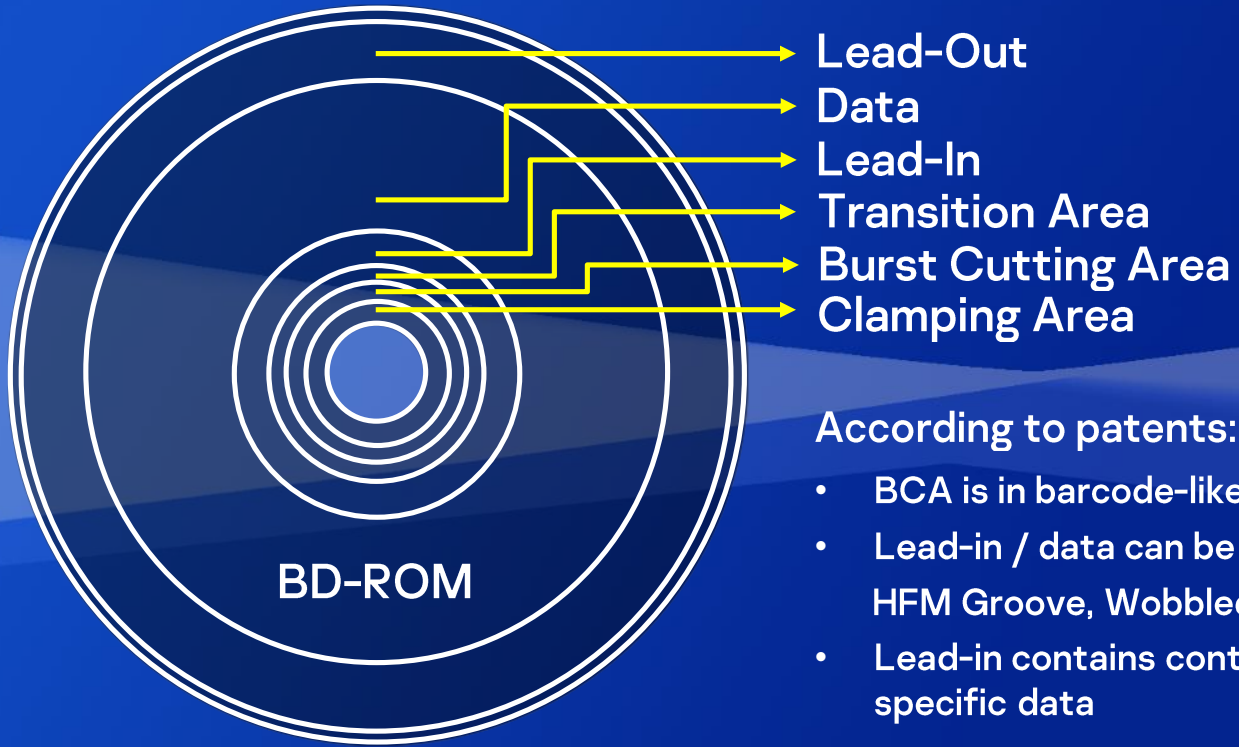
Better laser, better materials, better physical format

Very brief timeline:

- 2000: Sony unveils first Blu-ray disc prototypes
- 2002: Sony joins efforts with other industry partners
- 2006: Specifications were finalized
- 2006: First commercial BD-ROM player and first Blu-ray disc movies
- 2006: PlayStation 3 released
- 2013: PlayStation 4 released
- 2013: Xbox One released

Physical format is very well documented in whitepapers and patents

Three main types: Read-only (BD-ROM), Recordable (BD-R), and rewritable (BD-RE)



According to patents:

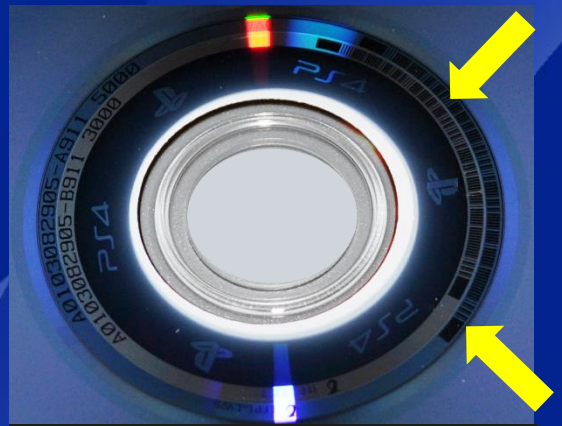
- BCA is in barcode-like format
- Lead-in / data can be recorded differently:
  HFM Groove, Wobbled Groove, Wobble Pit, Normal Pit
- Lead-in contains control information and application specific data

But patents do not reveal one simple thing – how PlayStation Blu-ray discs are verified?

In patents many things are application specific or described vaguely (e.g. BCA is in barcode format)

PS3 discs have only 1 visible barcode, but is it a clamping area?

PS4 discs have 2 barcodes in different format



There are also special PS3 and PS4 discs with 2 barcodes in same format

My thought was that drive firmware may reveal some details and secrets

PlayStation 3 Blu-ray drives

There have been a lot of different drive hardware revisions

If you will unpack PS3 FW UPDATE you will find firmware's for 12 (!) different drive models

Two of them seems to be for pre-retail units

First retail drive has a codename 302R and works over PATA interface

PCB of first PS3 retail drive. Main MCU is produced by Sony

PlayStation 3 Blu-ray drives (2)

After some time Sony decide to start using MCU's produced by Renesas
304R is a codename of first drive with Renesas MCU

After that Sony switched between Sony MCU and Renesas MCU for each new model:
302R, 303R, 306R, 310R, 314R – Drive models with Sony MCU
304R, 308R, 312R, 316R – Drive models with Renesas MCU
318R – The most recent model, not had a chance to check this one
With time PATA interface was replaced with SATA
In Slim revisions Blu-ray PCB's were integrated to motherboard

Sony was much more consistent with Blu-ray drives for PlayStation 4
If you will unpack PS4 FW UPDATE you will find firmware's for 6 different drive models

Sony finally abandoned their own MCU
402R, 406R, 408R, 410R, 412R – Drive models with Renesas MCU
420R - Drive model with MediaTek MCU, it come out just recently

PCB of first PS4 retail drive

Renesas is the most common chip for Blu-ray drives across PS3 and PS4

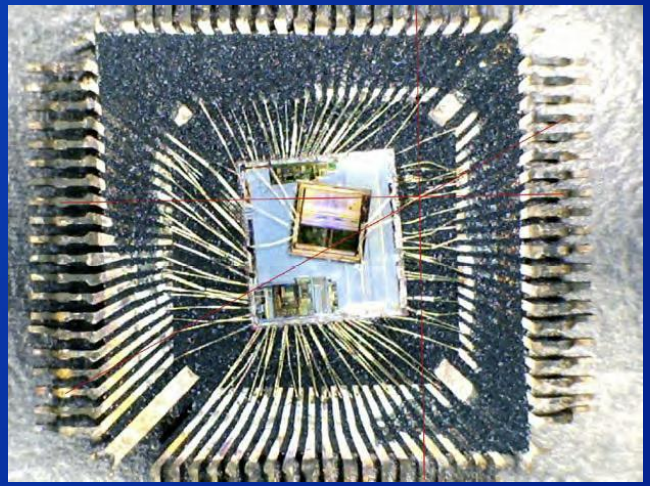
That's why its the main subject of this talk

Getting firmware out of Renesas MCU

Actually this technique came out from Xbox 360 scene

Quite often firmware is stored on flash chip that is a separate die inside a package

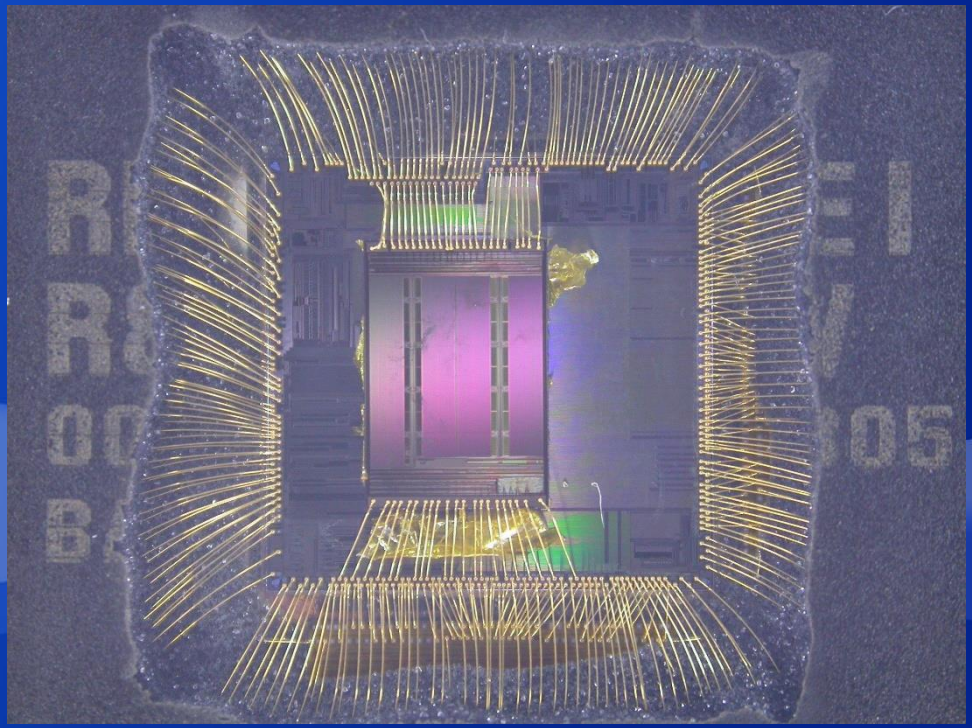
- Firmware is really large in size and separate die is easier for MCU manufacturer



Geremia's "kamikaze" hack for Xbox 360 drive

https://i.imgur.com/kYW08.jpg

Getting firmware out of Renesas MCU (2)

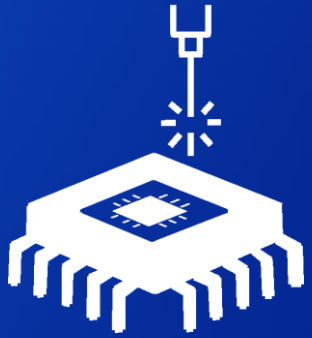


Decapsulated Renesas MCU of PS3 drive. Flash die is on top

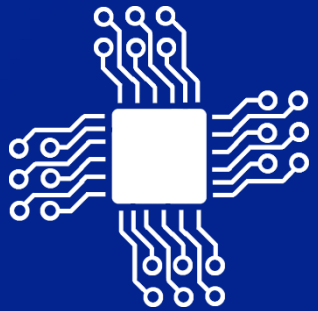
https://www.psdevwiki.com/ps4/File:Renesas_SCEI_r8j32...fpv_decapped.jpg

Getting firmware out of Renesas MCU (3)

Use acid to decapsulate MCU



Cut bond wires with laser



Rebond wires to custom PCB and read flash

Flash dump of Renesas MCU for PlayStation 3

Flash size is 2 MB

Contains: Mini FW (EMBOOT – Emergency Boot), Main FW, configuration data

```
3:0000h: 43 6F 70 79 72 69 67 68 74 28 43 29 20 32 30 30  Copyright(C) 200
3:0010h: 38 2D 32 30 31 31 20 53 6F 6E 79 20 43 6F 6D 70  8-2011 Sony Comp
3:0020h: 75 74 65 72 20 45 6E 74 65 72 74 61 69 6E 6D 65  uter Entertainme
3:0030h: 6E 74 20 49 6E 63 2E 20 20 20 20 20 20 20 20 20  nt Inc.
3:0040h: 00 00 00 01 00 19 00 00 20 4A 75 6E 32 30 2C 32  ........ Jun20,2
3:0050h: 30 31 31 20 31 31 3A 30 30 3A 30 30 20 20 00 00  011 11:00:00  ..
3:0060h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

What CPU architecture?

- Renesas bought semiconductor businesses of Hitachi, Mitsubishi and merged with NEC
- A variety of microcontrollers https://www.renesas.com/eu/en/products.html
- Google'd documents revealed that Blu-ray MCU's are based on H8S
- Firmware uses ldmac instruction, so it's a H8S/2600. Supported by IDA Pro

H8S/2600

Nice RISC-like architecture, reminds x86

69 instructions / 8 addressing modes

General registers: er0-er6, er7 is stack pointer

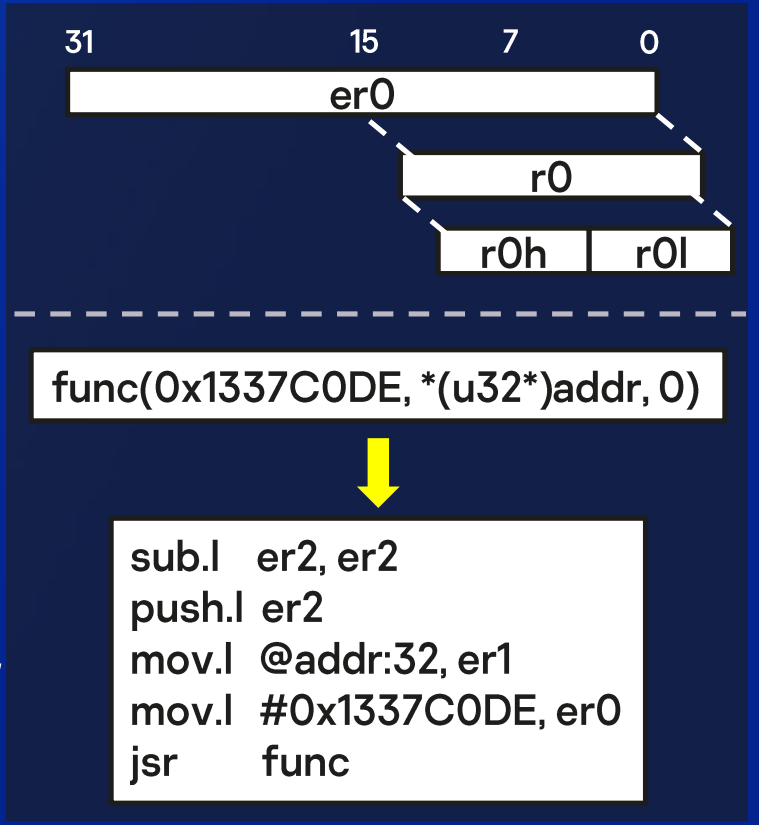
Compilers:

- Renesas Hi-performance Embedded Workshop (HEW)
- GCC
- IAR Embedded Workbench for H8

Each compiler uses different calling convention

There are even changes between different versions of HEW

Usually arguments are passed through er0, er1 and stack



Hardware manuals: https://www.renesas.com/us/en/products/microcontrollers-microprocessors/h8s.html

Most of space on flash is taken by main firmware (0x190000 bytes)

Do you remember that case when you reverse something and in each function there is a debug message that reveals name of a function? **Its not that case** ☹

There are only about 45 strings in whole firmware, I can even fit them on this slide

How do developers debug their code without debug messages?

- Basically 2 MB is barely enough to fit firmware, simply no space for debug strings
- Log trace functionality exists in firmware, but integer is passed as ID of a message
- ID is converted to a string inside special software that developer has
- Common technique, but bad for RE

You want to download as much as you can from website of hardware manufacturer

- Source code
- Libraries
- Compilers

You need it to make the process of reverse engineering easier:

- Compile sources and bindiff with your firmware
- Generate FLIRT signatures for IDA Pro
- Get MMIO definitions and structures

Renesas provides a huge list of stuff for download

Unfortunately any DVD/BD related stuff is not available publicly

- Its really complicated the whole project, but I am used to such things

Renesas owns a dozens of different RTOS'es, some are available for download

Renesas Hi-performance Embedded Workshop (HEW) is also available for download

- We get compiler that is likely was used to compile firmware

https://www.renesas.com/jp/ja/products/software-tools/discontinued-tools.html#downloads

Looking through files of HEW compiler I was not able to find sources and libraries

It appeared that they are stored inside .pak files: libsrc.pak, libinc.pak, asm2600a.pak, etc

```
0000h: 5A 48 00 00 11 79 00 00 34 39 00 09 61 64 64 64  ZH...y..49..addd
0010h: 33 2E 73 61 64 C7 3B 2F 2A F0 FF 02 0F 08 02 2F  3.sadÇ;/*ðÿ..../
0020h: 0D FD 0A EE F0 20 20 48 38 53 2C FF 48 38 2F 33  .ý.îð  H8S,ÿH8/3
0030h: 30 30 20 53 FF 45 52 49 45 53 20 43 20 FF 43 6F  00 SÿERIES C ÿCo
```

Example of asm2600a.pak

Necessary files are unpacked during compilation to temporary folder

Unpacking algorithm was found inside lbg38.exe - abbreviation for Library Generator

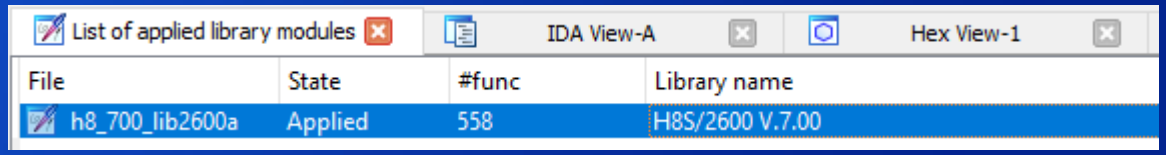
```
;---------------------------------------;
; H8S,H8/300 SERIES C/C++ Compiler Ver. 6.0 ;
; Runtime Library                       ;
; Copyright (C) 1994,2002 Hitachi,Ltd.  ;
; Licensed Material of Hitachi,Ltd.     ;
;---------------------------------------;
;
;Assemble:
;  asm38 -def=$CPU=<CPU> i_itoif
```

Example of unpacked assembly file

Reverse engineering compilers (Results)

HEW did not contain any useful information about hardware

But it was possible to generate IDA Pro FLIRT signatures



Functions: Bfinl - Set bit field on long, Divl - Divide long, Ftol - Transform float to long, etc...

They are used in firmware, so it's a useful finding

Reverse engineering real-time operating system (RTOS)

Finding RTOS functions is very important

Data and control flow might be passed between different tasks

Got many RTOS'es from Renesas

All of them are similar, but still have some differences

In most cases code is written in assembly for different architectures

Nothing closely matches RTOS used in drive firmware, so they are not really useful

The best thing about reverse engineering firmware developed by Japanese company is that it likely to have RTOS that follows µITRON (Micro Industrial TRON) specification

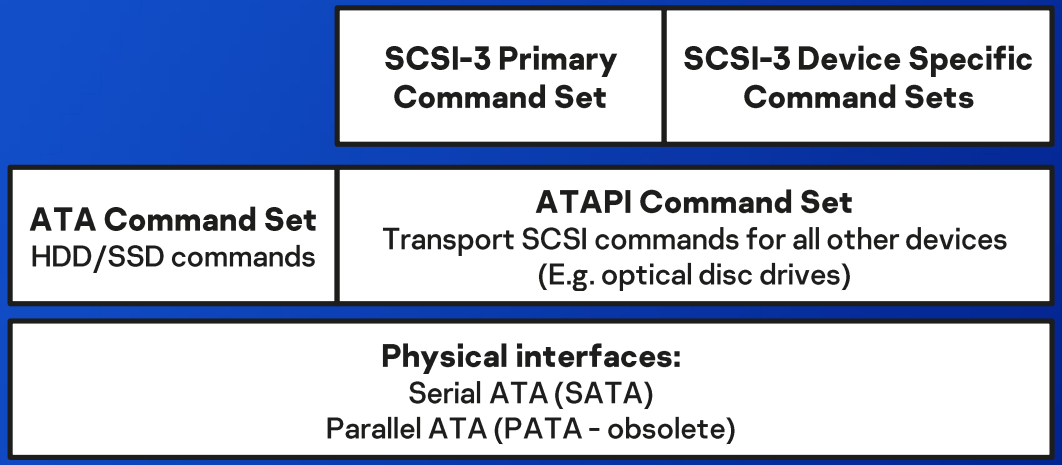
This specification defines names of functions, their arguments, etc (more than 300 pages)

µITRON specification: http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf

AT Attachment (ATA) interface

Hard disk drives (HDD), solid state drives (SSD), CD/DVD/BD optical disc drives

What they have in common?



ATA protocols hierarchy

For optical discs there are two competing specifications:

1) Multi-Media Commands (MMC)

2) “Mt. Fuji Commands for Multimedia Devices” by SFF committee, Pioneer, Panasonic, Sony, Toshiba

Reverse engineering ATAPI devices

SCSI-3 Primary Command Set implements INQUIRY command

It sends information about device to client

This information contains VENDOR IDENTIFICATION, PRODUCT IDENTIFICATION, etc

Information obtained from INQUIRY command

Look for such strings in firmware and you will find handler for SCSI commands

Then you can find interesting code and reverse it (with help of specifications)

Embedded device reverse engineering roadmap

- ✅ Acquiring firmware
- ✅ Recognizing CPU architecture
- ❌ Collecting information about hardware
  - ❌ Debug strings
  - ❌ MMIO addresses definitions, names and structures
- ✅ Identifying standard library functions
- ✅ Identifying RTOS functions
- ✅ Pinpointing interesting logic and reverse engineering it
- ⬜ Emulating firmware to aid in static analysis
- ⬜ Getting code execution (as a final goal or a way to enable field for experiments)

Emulating firmware

GDB includes a simulator for H8 architecture

./configure --target=h8300-hitachi-elf --enable-sim

Convert firmware to ELF and you are good to go, can emulate some snippets of code

I actually like using IDA Pro as debugger GUI, but it has some flaws of course

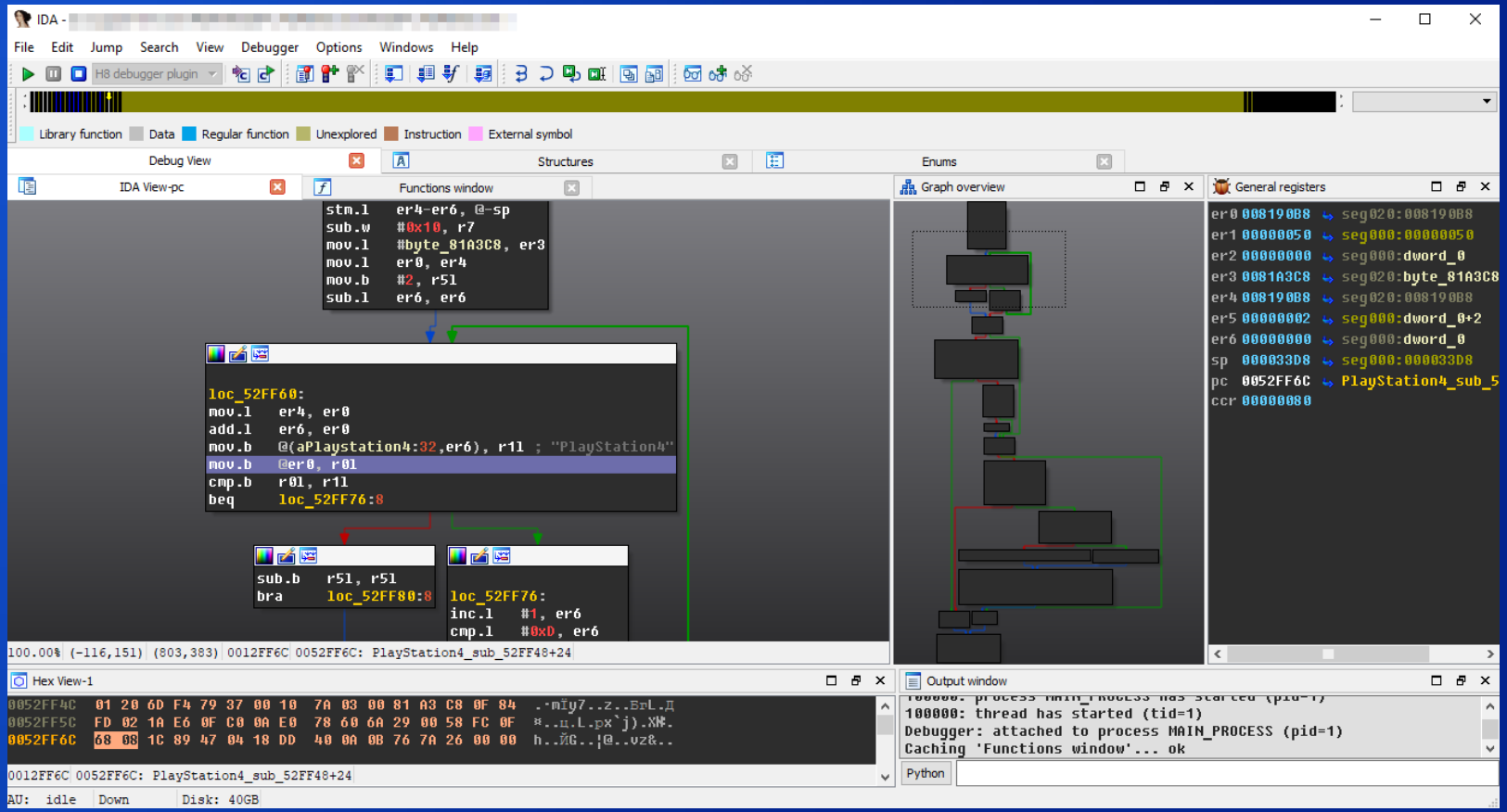
GDB debugger plugin that comes with IDA Pro is closed source

Recently Hex-Rays improved it a lot, but back then it was buggy and supported only a few targets

I wrote my own GDB debugger plugin to work with IDA Pro

It did not took much time to make, but it saved a lot of time while debugging this firmware and other officially not supported targets (e.g. GDB support for x64 targets was added only in IDA Pro 6.9)

Emulating firmware (In action)



Renesas Blu-ray drives for PC

While I was reverse engineering firmware for PlayStation 3, I actually was looking at multiple FW's

It appeared that there exist some Blu-ray drives for PC with Renesas MCU

Manufactured by Hitachi-LG Data Storage (HLDS / HL-DT-ST)

Not encrypted firmware can be grabbed from firmware update utility

PC and PS firmware's are different, but its clear that they are built using the same SDK

- Many Blu-ray/hardware related functions are the same
- Peripheral devices are the same and mapped to the same MMIO addresses
- PC firmware uses the same cryptographic processor
- PC firmware uses RX850 RTOS
- PC firmware contains a little more debug strings
  - Reveals the codename of Renesas Blu-ray LSI – "Indigo3"

Cryptographic processor

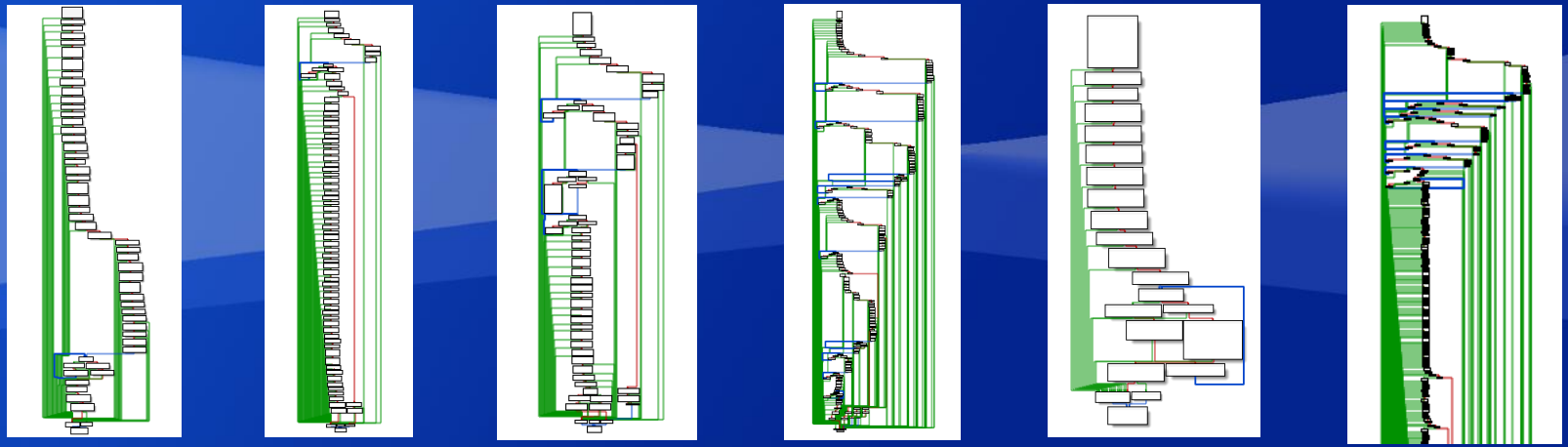
Huge part of firmware is occupied by crypto-related functions

All cryptographic operations are handled by specially dedicated IC

Crypto processor effectively protects all secrets

- You cant just dump firmware and reverse engineer all of it

Communication process with crypto processor is really complicated and obscure



Examples of graphs for crypto processor related functions

Cryptographic processor (2)

Crypto processor runs some kind of firmware

Functionality can be extended with additionally loaded modules (or patches) and additional keys

- PlayStation firmware uses these features

For me its not exactly clear why so many things are performed within main firmware

- Crypto processor has limited memory resources?

To play with crypto processor and test ideas we need to get code execution

```
if(!wait_0xFF3FF058()) goto done;

*(unsigned int*)0xFF3FF0C4 = 0x34E0A89;

if(!wait_0xFF3FF001_bit_6()) goto done;

*(unsigned int*)0xFF3FF0C0 = 0xD4398114;

if(!wait_0xFF3FF0CB()) goto done;

*(unsigned int*)0xFF3FF084 = 0x18;
*(unsigned int*)0xFF3FF0C4 = 0x410E0309;
*(unsigned int*)0xFF3FF004 = 0x40001;

if(!wait_0xFF3FF001_bit_6()) goto done;

set_key_to_0xFF3FF0C0((unsigned char*)key_0);

*(unsigned int*)0xFF3FF0C4 = 0x22450B19;

if(!wait_0xFF3FF001_bit_6()) goto done;

*(unsigned int*)0xFF3FF0C0 = 0x1D385210;
```

Snippet from crypto-related function

Exploiting ATAPI devices

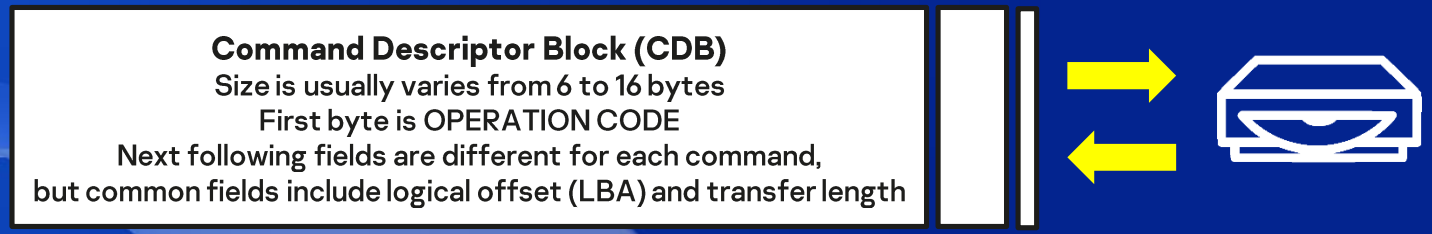
Earlier this year I gave a presentation at CanSecWest – “Hacking Microcontroller Firmware through a USB”

More details here: https://securelist.com/hacking-microcontroller-firmware-through-a-usb/

In this research I examined how awesome protocol of control transfers is for USB exploitation

SCSI protocol might be even more awesome, though less common

Client sends a Command Descriptor Block (CDB) to device



Depending on type of CDB, commands can be used to read and write data from and to device

Device also provides status of command and error code (sense) in response to client

It seems that firmware itself was developed by some 3rd party company

Then when it was ready if was handled to Sony, to add console specific stuff

All general SCSI commands are looking fine

But some commands implemented by Sony doesn't seem to have boundary checks

One of the examples is vulnerable command with OPERATION CODE = 0xE1

This command is used for authentication of Blu-ray drive and video game console

So the main purpose of it is to implement security …

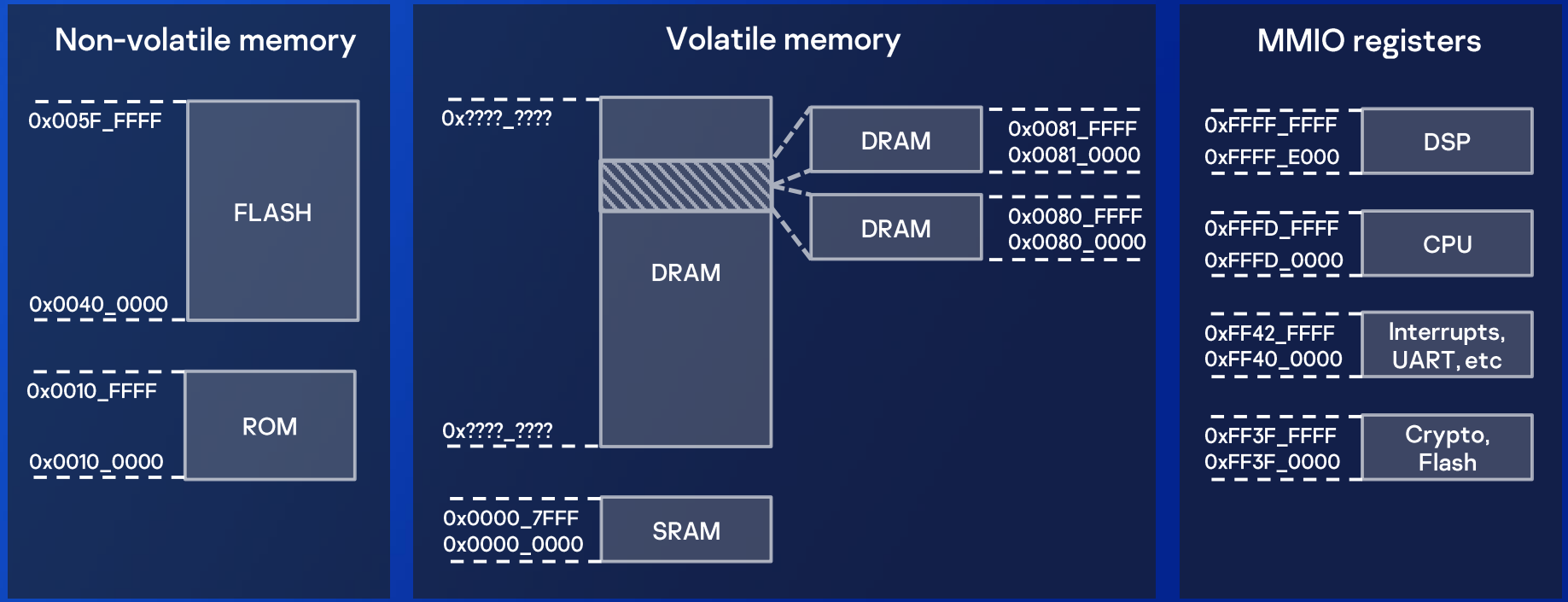
Byte	CDB Fields
0	OPERATION CODE (0xE1)
1	Transfer length
2	
…	…

OOB write

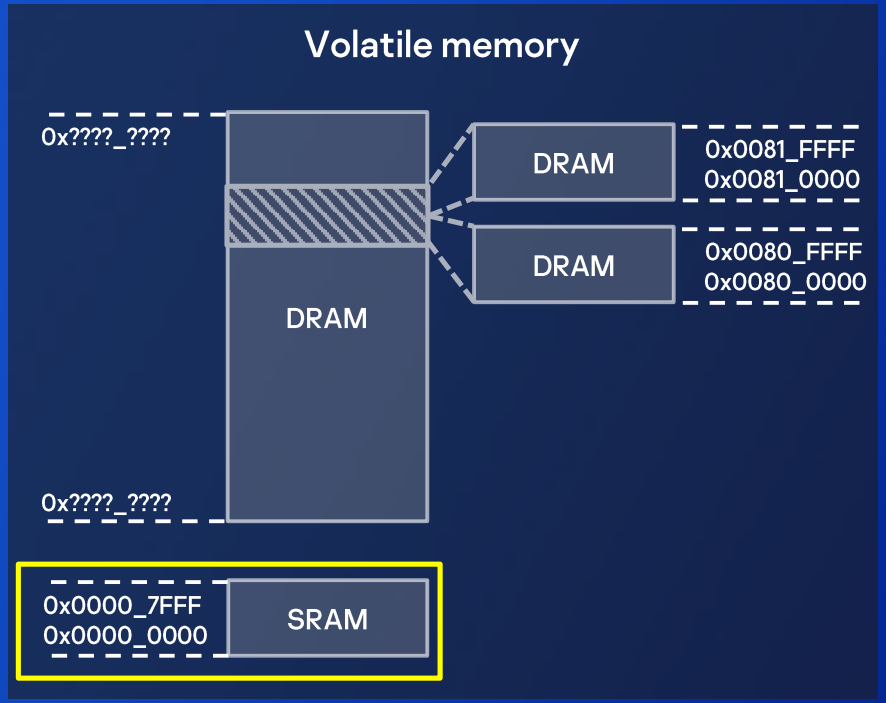
Buffer size is 0x80 bytes, bug lets to overwrite up to 0xFFFF bytes

Static buffer is located somewhere in the middle of 0x81XXXX address space

Memory map


Non-volatile memory
0x005F_FFFF
FLASH
0x0040_0000
0x0010_FFFF
ROM
0x0010_0000
Volatile memory
0x????_????
DRAM
0x0081_FFFF
0x0081_0000
DRAM
0x0080_FFFF
0x0080_0000
DRAM
0x????_????
0x0000_7FFF
0x0000_0000
SRAM
MMIO registers
0xFFFF_FFFF
0xFFFF_E000
DSP
0xFFFD_FFFF
0xFFFD_0000
CPU
0xFF42_FFFF
0xFF40_0000
Interrupts, UART, etc
0xFF3F_FFFF
0xFF3F_0000
Crypto, Flash

Volatile memory (SRAM)



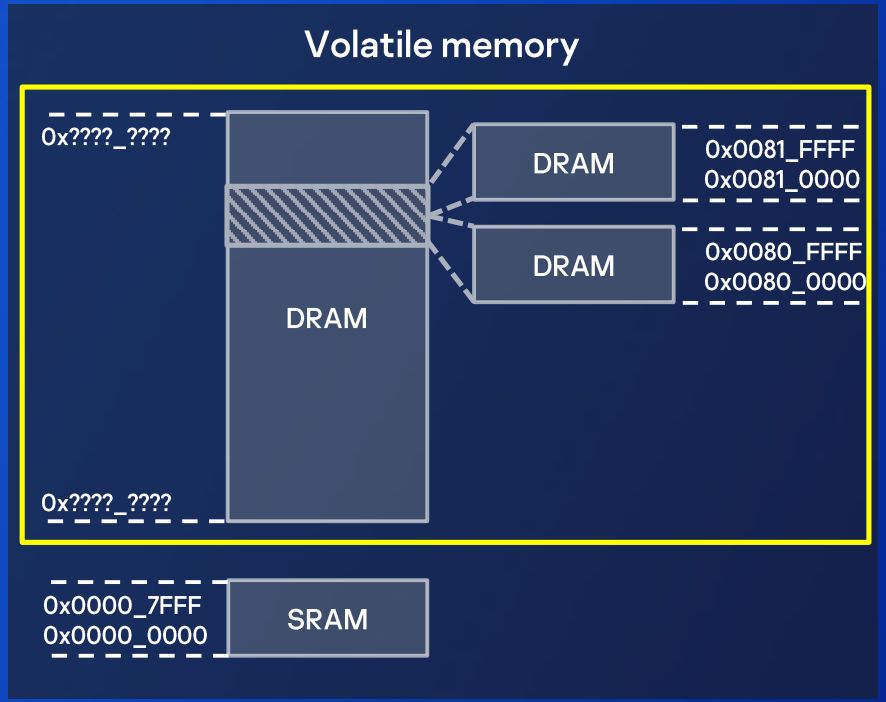
SRAM - Static random access memory

- Small size (0x8000 bytes)
- Executable (Configurable)

Contains:

- Exception / Interrupt vector table
- Code of real-time operating system
- Important variables, pointers
- Task stacks

Volatile memory (DRAM)



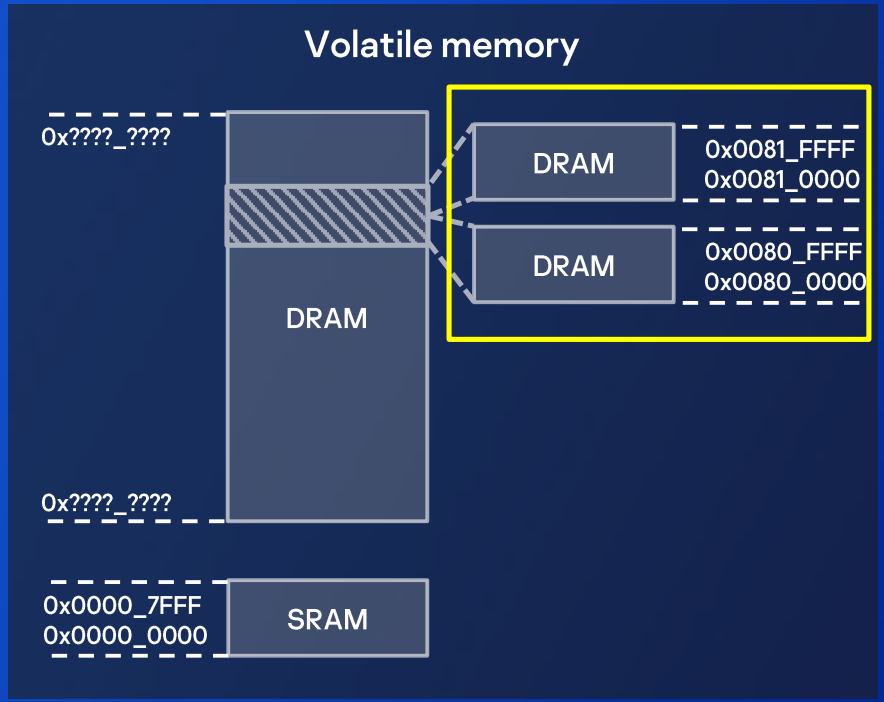
DRAM - Dynamic random access memory

- Large size (Several megabytes)
- Initially exact memory location and size were unknown because most of the time DRAM is accessed through Direct Memory Access (DMA)

Contains:

- Data from disc
- Data from SCSI client
- Data that do not fit to SRAM

Volatile memory (DRAM) (2)



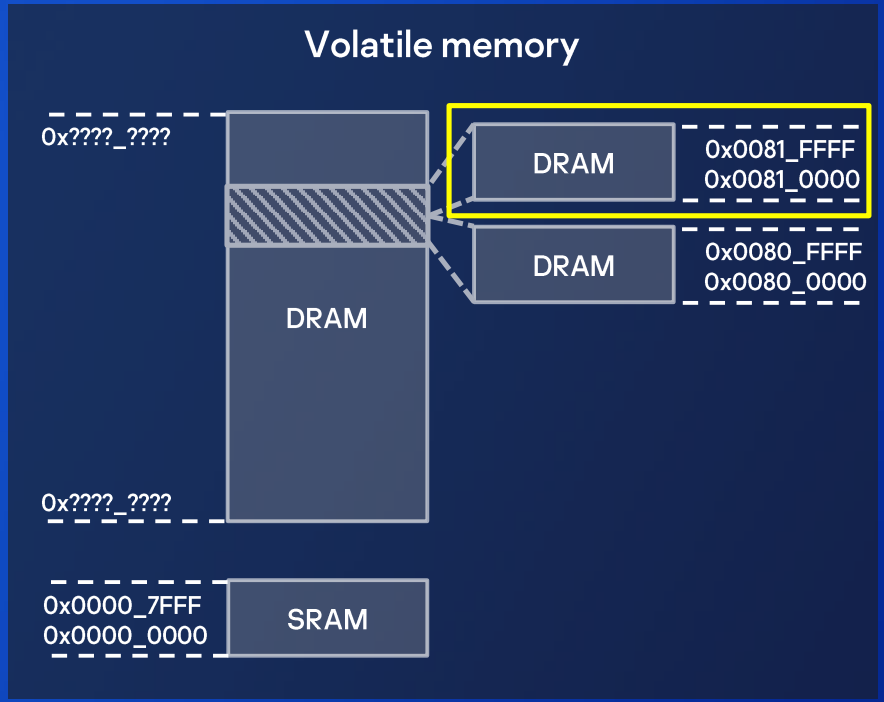
Firmware is too big, SRAM is too small to fit all variables

Staff that didn't fit to SRAM is stored in special regions of DRAM

PS firmware actively uses memory region 0x0081_0000

Memory region 0x0080_0000 is unused, but I found out that it exist from Hitachi-LG Data Storage firmware

Volatile memory (DRAM) (3)



We are able to overwrite 0x0081_0000 memory region with out of bounds write

How to exploit that?

Exploitation turned out to be a very difficult

All variables are located at static addresses

- Heap exploitation techniques are not working there

At first you need to find a very good exploitation primitive

You need to overwrite many variables prior to reaching this primitive without crashing the device

- Debugging is complicated, need to emulate/understand large portions of firmware

I ended up reverse engineering all functions that access data in 0x0081_0000 memory region

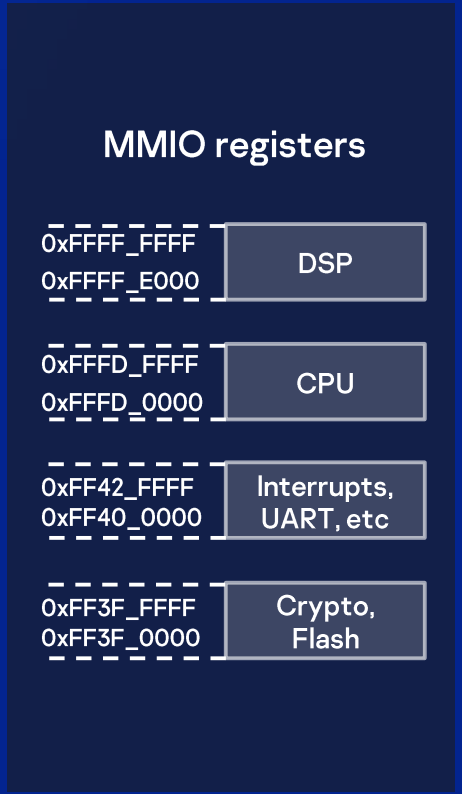
- There are no virtual function pointers stored there (no good candidates for overwrite)
- There are a lot of buffers, structures, other variables… but not that much pointers
- Eventually I was able to find good exploitation primitives

Stumble into new bug class

I never had a chance to finish this exploit actually

While writing this exploit I decided to take a look at MMIO regs

In result I have found a new source for vulnerabilities



DSP registers are very interesting

They are responsible for most of disc drive related functionality:

- ATAPI interface
- Laser
- Servo
- Disc data de-modulation (BCA, PIC, AACS Volume ID, etc)
- DSP firmware loading
- Etc

It would be so nice if we had access to it...

Actually we do!

The whole area 0xFFFF_E000-0xFFFF_FFFF is available for R/W through special SCSI commands

Read Indigo DSP registers

Byte	CDB Fields	
0	OPERATION CODE (0x3C)	
1	Reserved	Mode (1)
2	Buffer ID (0x40)	
3	Offset	
4		
5	Length	
6		
...	...	

Write Indigo DSP registers

Byte	CDB Fields	
0	OPERATION CODE (0x3B)	
1	Reserved	Mode (2)
2	Buffer ID (0x40)	
3	Offset	
4		
5	Length	
6		
...	...	

Accessing Indigo DSP registers (2)

Most likely these functions are a part of standard diagnostic functionality

The same functions are available in Hitachi-LG Data Storage firmware

Read Indigo DSP registers

Byte	CDB Fields	
0	OPERATION CODE (0x3C)	
1	Reserved	Mode (5)
2	Buffer ID (4)	
3	Offset	
4		
5	Length	
6		
...	...	

Write Indigo DSP registers

Byte	CDB Fields	
0	OPERATION CODE (0x3B)	
1	Reserved	Mode (5)
2	Buffer ID (4)	
3	Offset	
4		
5	Length	
6		
...	...	

Direct Memory Access (DMA) registers

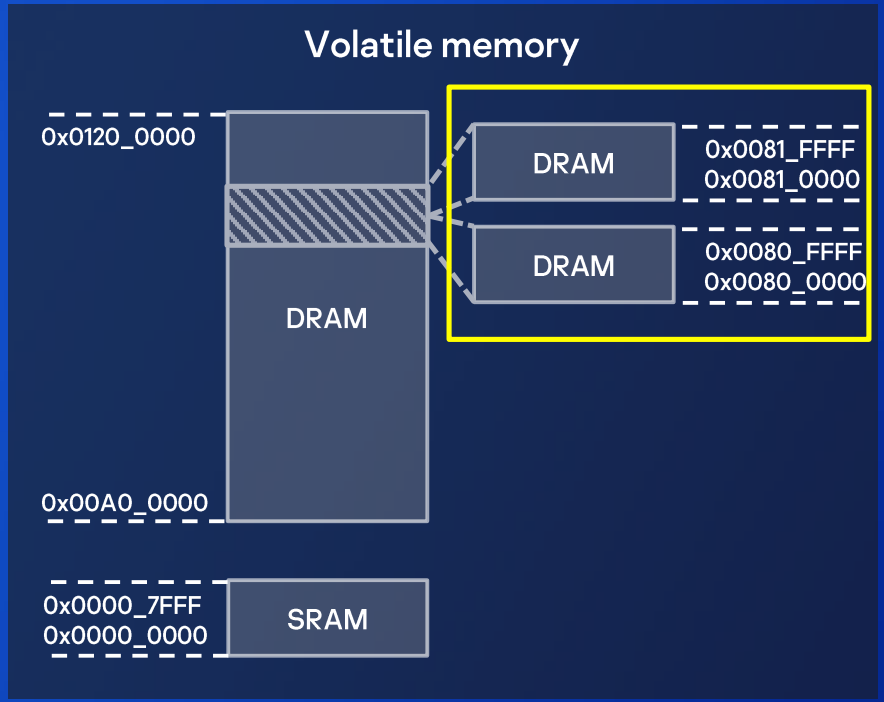
Do you remember when I said that DRAM is accessed mostly through DMA?

Several registers available in 0xFFFF_E000-0xFFFF_FFFF area are responsible for DMA

Firmware uses them to:

- Copy data from and into DRAM
- Map DRAM offsets to memory addresses

DRAM memory mappings



It appeared that:

- DRAM actually located at address 0x00A0_0000 (size: 8 MB)
- Special regions of DRAM are mapped using DSP registers

DRAM memory mappings (2)

PS firmware uses 4 groups of memory mapping registers

Mem1 registers:

0xFFFFFB00 – Region (always set to 0xF8)

0xFFFFFB28 Bit 0 - First/Second half of DRAM

0xFFFFFB04 Bit 0 - Enable/Disable

Mem2 registers:

0xFFFFFB01 – Region (always set to 0xF0)

0xFFFFFB29 Bit 0 - First/Second half of DRAM

0xFFFFFB04 Bit 1 - Enable/Disable

. . .

Each group of registers maps 0x20000 bytes from DRAM offset to predefined address

Mem1 and Mem2 mappings are set during initialization, Mem3 and Mem4 are set by different functions

Mem1 address is 0x800000, firmware maps it to DRAM offset 0x7E0000

Mem2 address is 0x820000, firmware maps it to DRAM offset 0x7C0000

DRAM offset is calculated like this: Region value is multiplied by 0x4000, added to 0 or 0x400000

Read/Write DRAM through DSP registers

Read 4 bytes of DRAM

```
*(u8*)0xFFFFFB48 = 1;

*(u32*)0xFFFFFB50 = Address;

*(u8*)0xFFFFFB4A = 0x40;
*(u8*)0xFFFFFB4A = 0xC0;

Value = *(u32*)0xFFFFFB54;
```

Write 4 bytes to DRAM

```
*(u8*)0xFFFFFB48 = 0;

*(u32*)0xFFFFFB50 = Address;

DMA_CNTRL = *(u8*)0xFFFFFB4A;
*(u8*)0xFFFFFB4A = DMA_CNTRL & 0x8F;

*(u32*)0xFFFFFB54 = Value;

DMA_CNTRL = *(u8*)0xFFFFFB4A;
*(u8*)0xFFFFFB4A = (DMA_CNTRL & 0xF0) | 0xF;
DMA_CNTRL = *(u8*)0xFFFFFB4A;
*(u8*)0xFFFFFB4A = (DMA_CNTRL & 0x7F) | 0x80;
```

We have access to DMA registers with SCSI commands

Are we able to manipulate contents of DRAM?

Multiple ways to achieve this:

1) We are able to remap Mem1 to another DRAM offset and use OOB exploit to write there

- May lead to undefined behaviors
- Debugging is complicated

2) We are able to use DMA registers to write our data directly

- Actually not, because these registers are in use by firmware

Testing on real hardware

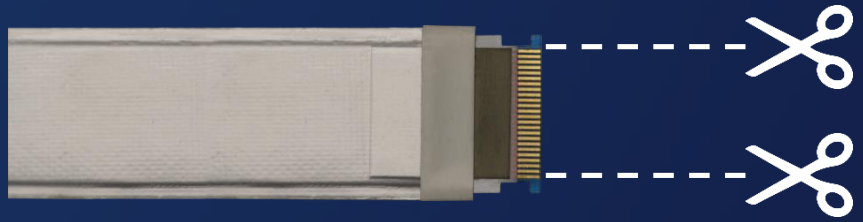
After finding vulnerabilities come a time to test ideas and develop exploit

There are two ways:

1) Jailbreak a console and install Linux to communicate with Blu-ray drive
2) Disconnect Blu-ray drive from console and connect to PC

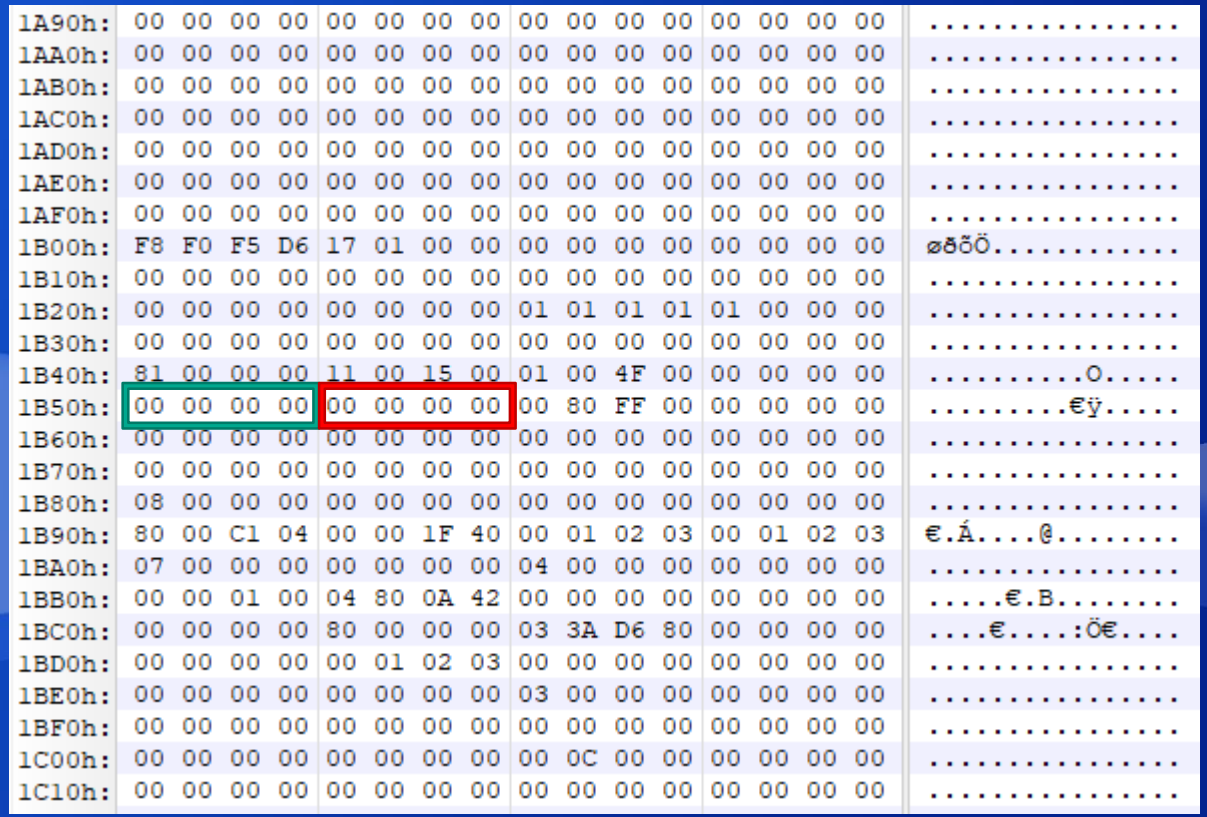
I decided to take a second route as its more convenient

Luckily it was quite easy to find and buy a ready made solution for doing just that



Public solution supported only PS3 FFC cables
I had to modify PS4 FFC cable with scissors

DSP registers dump



1A90h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AA0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AB0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AC0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AD0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AE0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1AF0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B00h:	F8	F0	F5	D6	17	01	00	00	00	00	00	00	00	00	00	00	øðõÖ............
1B10h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B20h:	00	00	00	00	00	00	00	00	01	01	01	01	01	00	00	00	....
1B30h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B40h:	81	00	00	00	11	00	15	00	01	00	4F	00	00	00	00	00	..........O.....
1B50h:	00	00	00	00	00	00	00	00	00	80	FF	00	00	00	00	00	.........€ÿ.....
1B60h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B70h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B80h:	08	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1B90h:	80	00	C1	04	00	00	1F	40	00	01	02	03	00	01	02	03	€.Á....@........
1BA0h:	07	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	....
1BB0h:	00	00	01	00	04	80	0A	42	00	00	00	00	00	00	00	00	.....€.B........
1BC0h:	00	00	00	00	80	00	00	00	03	3A	D6	80	00	00	00	00	....€....:Ö€....
1BD0h:	00	00	00	00	00	01	02	03	00	00	00	00	00	00	00	00	....
1BE0h:	00	00	00	00	00	00	00	00	03	00	00	00	00	00	00	00	....
1BF0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
1C00h:	00	00	00	00	00	00	00	00	00	0C	00	00	00	00	00	00	....
1C10h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

DMA address

DMA value

Pleasant surprise

After dumping DSP registers I found out that DMA registers where not in use!

In newer models Sony stopped to use DMA registers to access DRAM

They just use absolute memory addresses

It means that we have R/W access to the whole DRAM!

Doing cool stuff with full access to DRAM

Usually DRAM is full of disc data

But when there is no disc inside, there are a lot of unused space

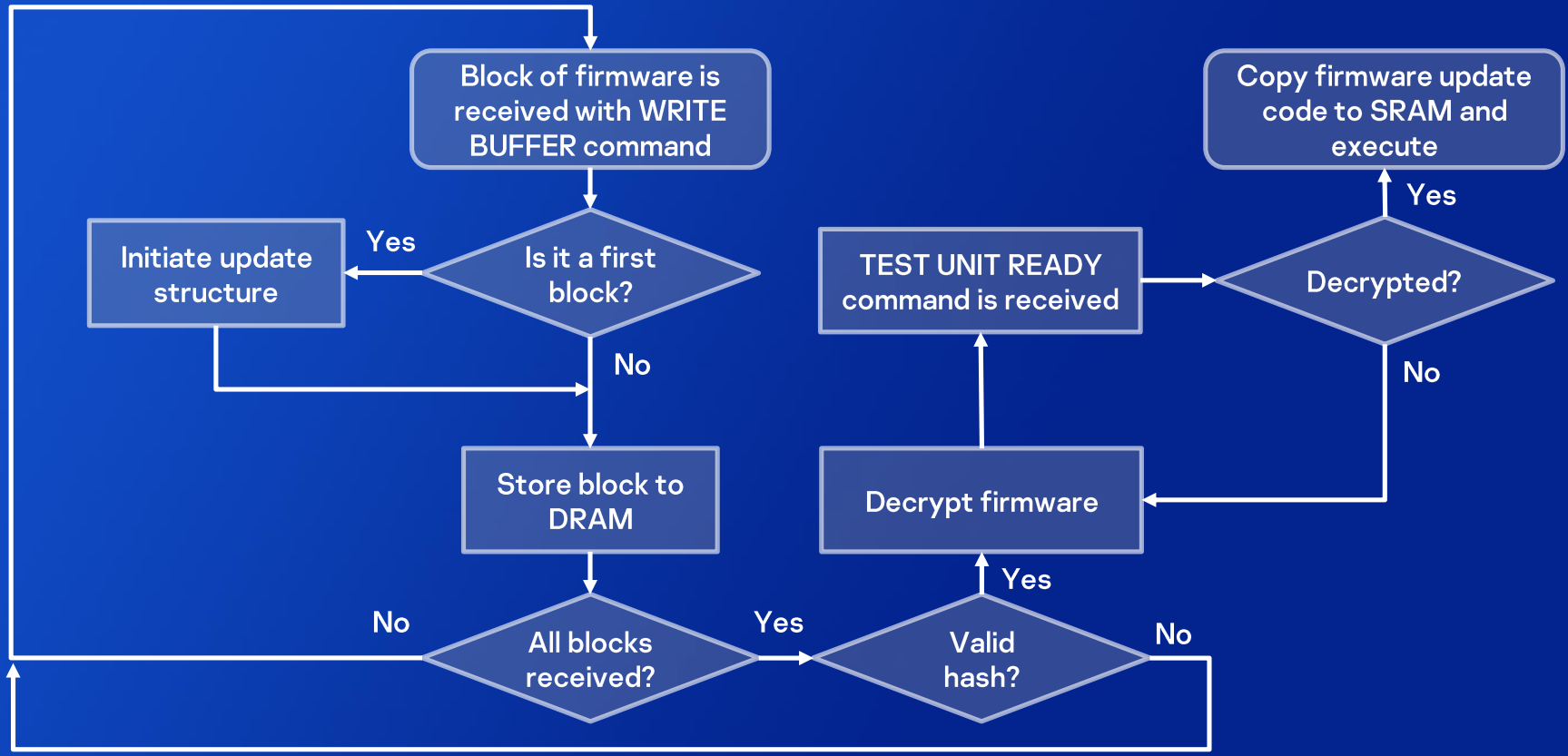
And this space is used to store firmware during update

The procedure of Blu-ray drive firmware update for PS3 is well known and documented here:

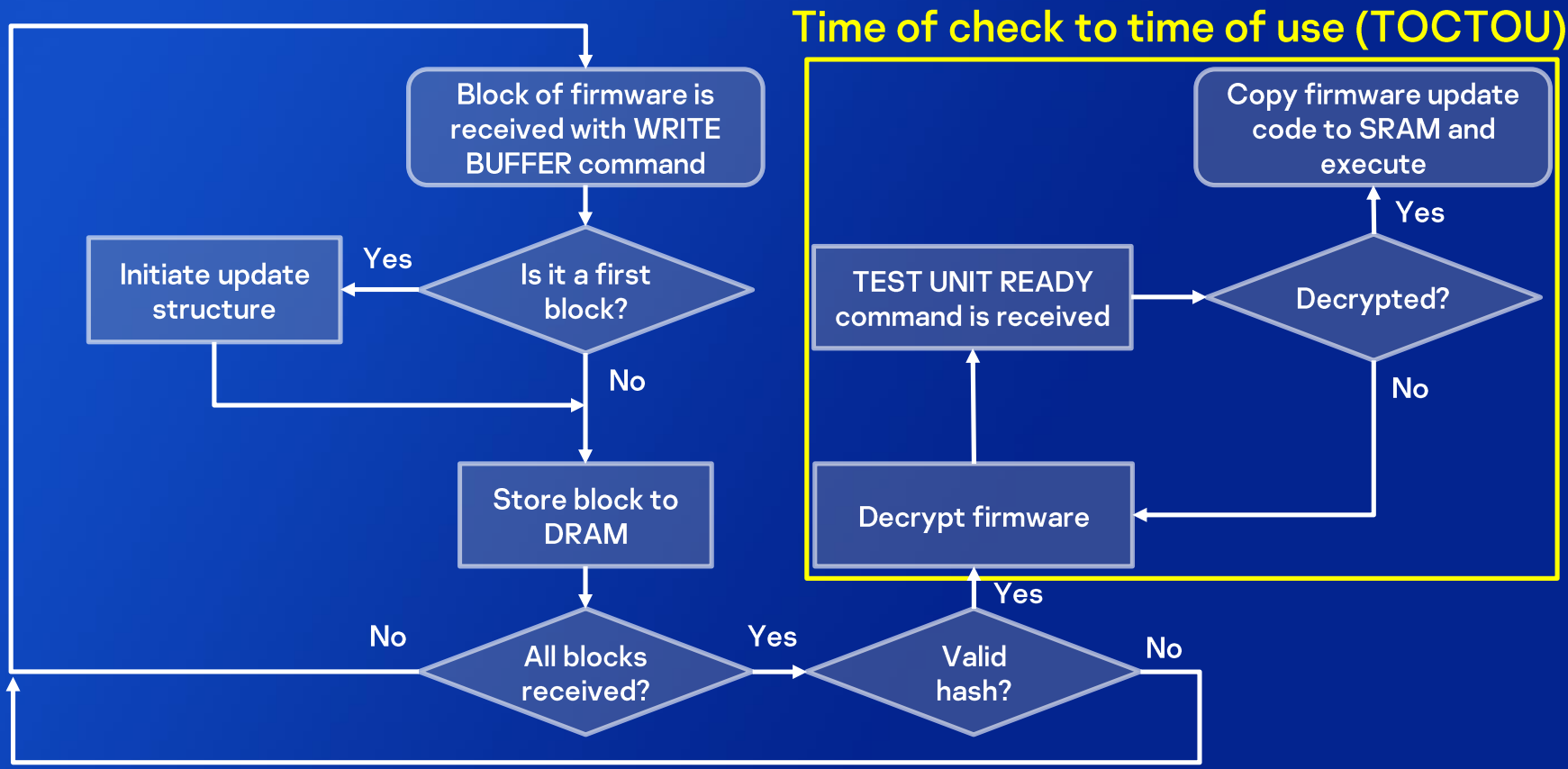
https://www.psdevwiki.com/ps3/Updating_Bluray_Drive_Firmware_on_Linux

The procedure of firmware update is exactly the same for PS3 and PS4

Doing cool stuff with full access to DRAM (2)

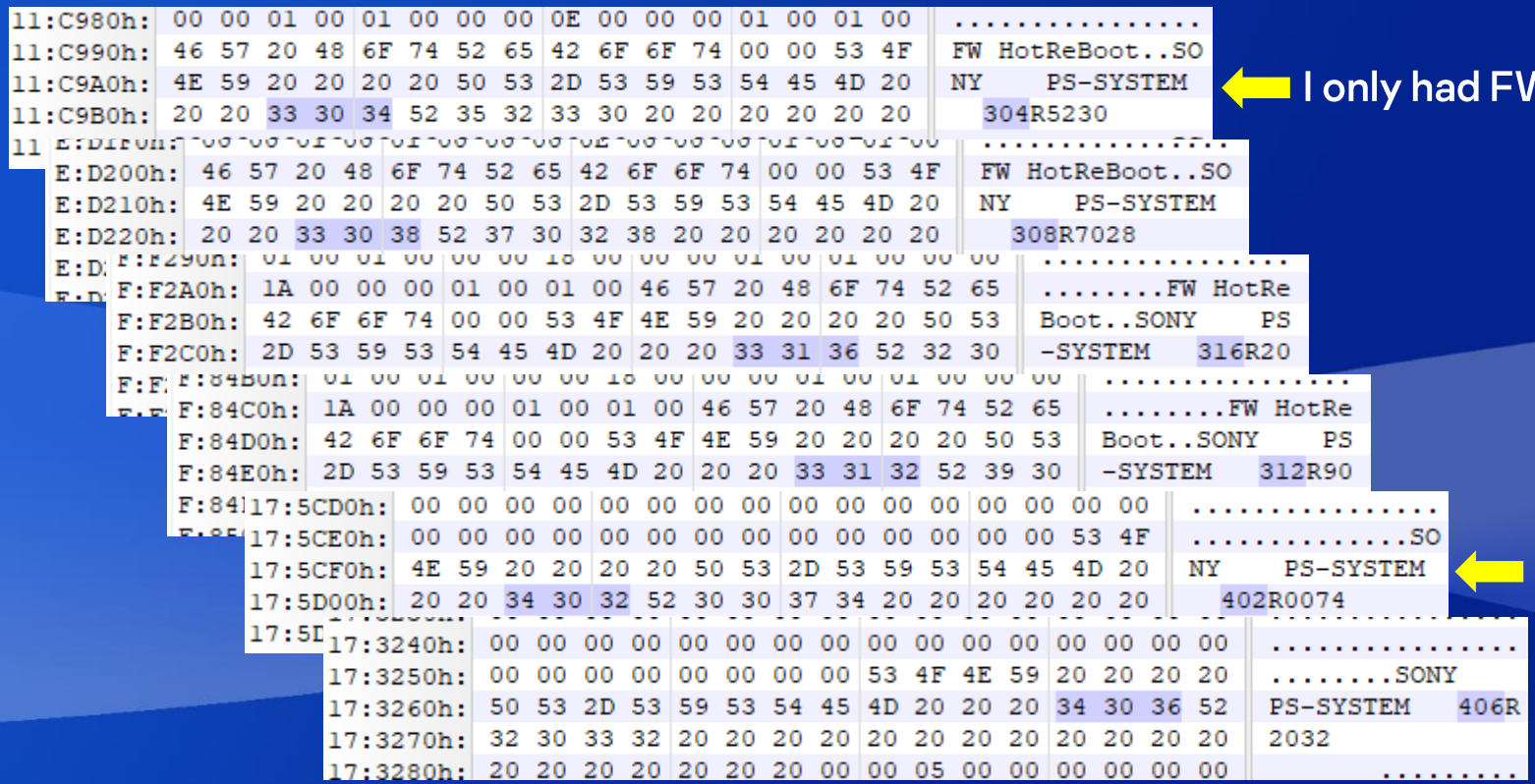


Doing cool stuff with full access to DRAM (3)



Decrypting firmware with the oracle

We can send firmware to drive, wait until its decrypted, dump decrypted firmware from DRAM



Code execution

On this stage manipulating firmware image to get code execution is trivial

All update structures are stored in DRAM

When you exploit such devices you need to be extremely careful

If you do something wrong you can brick device without options to restore it

I verified everything many times and my exploit worked at first attempt

But I need to mention that EMBOOT (Emergency Boot) firmware exist

If hash of main firmware fails on boot, EMBOOT firmware will be loaded and you can unbrick

Why did it happen?

Most likely scenario:

- When firmware was handled to Sony, engineers didn't really understood functionality that is available trough DSP registers
- SCSI commands to read and write DSP registers were left for diagnostic purposes
- Security risks represented by free use of DSP registers were not considered

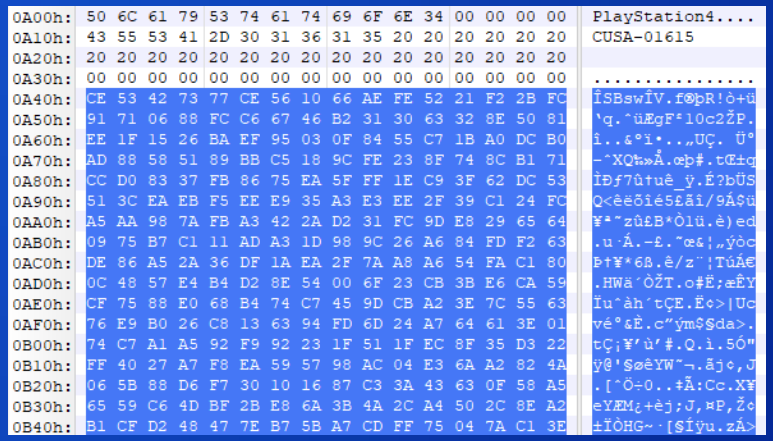
Mystery disc data

With code execution comes a time for experiments

When PS3 and PS4 disc are mastered at factory a special blob is added to fixed offset

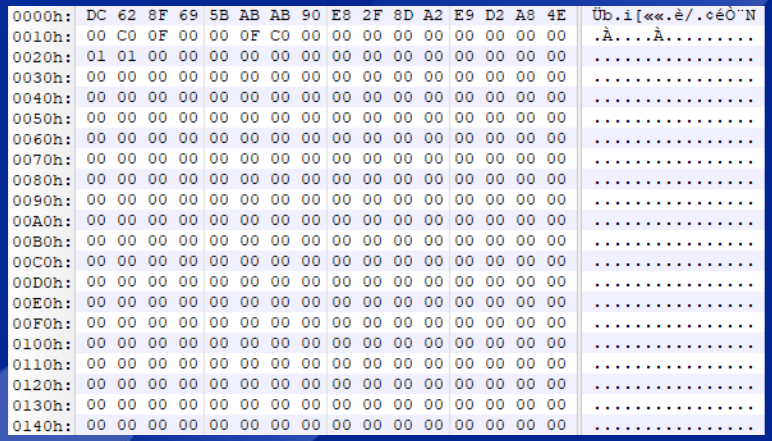
This blob was a mystery for community because code that accesses it was nowhere to be found

These blobs are signed and encrypted, code execution lets to take a look inside



0A00h:	50	6C	61	79	53	74	61	74	69	6F	6E	34	00	00	00	00	PlayStation4....
0A10h:	43	55	53	41	2D	30	31	36	31	35	20	20	20	20	20	20	CUSA-01615
0A20h:	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
0A30h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0A40h:	CE	53	42	73	77	CE	56	10	66	AE	FE	52	21	F2	2B	FC	ÎSBswÎV.f®þR!ò+ü
0A50h:	91	71	06	88	FC	C6	67	46	B2	31	30	63	32	8E	50	81	'q.ˆüÆgF²10c2ŽP.
0A60h:	EE	1F	15	26	BA	EF	95	03	0F	84	55	C7	1B	A0	DC	B0	î..&°ï•..„UÇ. Ü°
0A70h:	AD	88	58	51	89	BB	C5	18	9C	FE	23	8F	74	8C	B1	71	-ˆXQ‰»Å.œþ#.tŒ±q
0A80h:	CC	D0	83	37	FB	86	75	EA	5F	FF	1E	C9	3F	62	DC	53	ÌÐƒ7û†uê_ÿ.É?bÜS
0A90h:	51	3C	EA	EB	F5	EE	E9	35	A3	E3	EE	2F	39	C1	24	FC	Q<êëõîé5£ãî/9Á$ü
0AA0h:	A5	AA	98	7A	FB	A3	42	2A	D2	31	FC	9D	E8	29	65	64	¥ª˜zû£B*Ò1ü.è)ed
0AB0h:	09	75	B7	C1	11	AD	A3	1D	98	9C	26	A6	84	FD	F2	63	.u·Á.-£.˜œ&¦„ýòc
0AC0h:	DE	86	A5	2A	36	DF	1A	EA	2F	7A	A8	A6	54	FA	C1	80	Þ†¥*6ß.ê/z¨¦TúÁ€
0AD0h:	0C	48	57	E4	B4	D2	8E	54	00	6F	23	CB	3B	E6	CA	59	.HWä´ÒŽT.o#Ë;æÊY
0AE0h:	CF	75	88	E0	68	B4	74	C7	45	9D	CB	A2	3E	7C	55	63	Ïuˆàh´tÇE.Ë¢>|Uc
0AF0h:	76	E9	B0	26	C8	13	63	94	FD	6D	24	A7	64	61	3E	01	vé°&È.c”ým$§da>.
0B00h:	74	C7	A1	A5	92	F9	92	23	1F	51	1F	EC	8F	35	D3	22	tÇ¡¥'ù'#.Q.ì.5Ó"
0B10h:	FF	40	27	A7	F8	EA	59	57	98	AC	04	E3	6A	A2	82	4A	ÿ@'§øêYW˜¬.ãj¢‚J
0B20h:	06	5B	88	D6	F7	30	10	16	87	C3	3A	43	63	0F	58	A5	.[ˆÖ÷0..‡Ã:Cc.X¥
0B30h:	65	59	C6	4D	BF	2B	E8	6A	3B	4A	2C	A4	50	2C	8E	A2	eYÆM¿+èj;J,¤P,Ž¢
0B40h:	B1	CF	D2	48	47	7E	B7	5B	A7	CD	FF	75	04	7A	C1	3E	±ÏÒHG~·[§Íÿu.zÁ>

Encrypted



0000h:	DC	62	8F	69	5B	AB	AB	90	E8	2F	8D	A2	E9	D2	A8	4E	Üb.i[««.è/.¢éÒ¨N
0010h:	00	C0	0F	00	00	0F	C0	00	00	00	00	00	00	00	00	00	.À....À.....
0020h:	01	01	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0030h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0040h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0050h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0060h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0070h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0080h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0090h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00A0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00B0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00C0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00D0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00E0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00F0h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0100h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0110h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0120h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0130h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0140h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Decrypted

Disc keys

It was interesting to take a look on how keys are obtained from disc

This information should be somehow related to the way how discs are verified

For PS3 there are 2 disc keys:

- First key is used for decryption of disc data
- Second key is used for encryption of save data

All of this is about the same for PS4

I found out that disc keys are returned from crypto processor, but only after drive authentication

I thought that the logic to read and construct disc keys is located inside crypto processor

Authentication

Drive authentication and drive's crypto processor are the main things behind ODD security

Drive authentication is secure and performed with per-console keys

I know only two ways to get per-console keys:

- Hack crypto processor of video game console (SPU for PS3 / SAMU for PS4)
- Hack crypto processor of Blu-ray drive

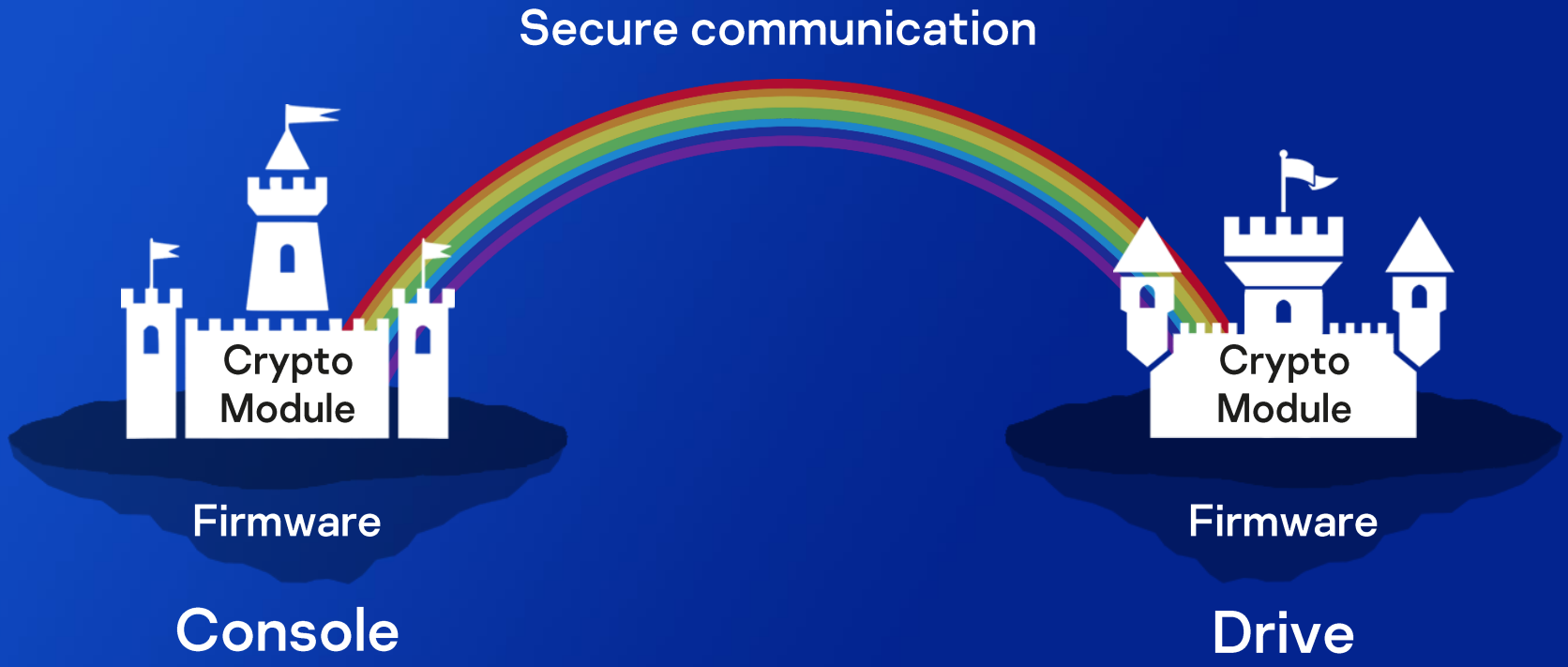
Its very hard to achieve such hacks and security model is effective against widespread piracy

Much more simple ways to pirate games always existed

If you hack main firmware of PlayStation - you **can** pirate games

If you hack firmware of PlayStation Blu-ray drive – you **can't** pirate games

PlayStation security model (Simplified)



Experiments with cryptographic processor

Code execution allowed me to play around with crypto processor

1) I was able to load crypto firmware from PS3 drive to PS4 drive

   Crypto processor of PS4 started to behave exactly like it should for PS3

   Even offsets of some MMIO registers have changed

2) As communication between main FW and crypto FW is complicated I decided to write a fuzzer

   If you change any value crypto processor will return error ☹

   After few errors crypto processor hangs and you need to reset a device

Cryptographic processor (3)


```
if(!wait_0xFF3FF058()) goto done;

*(unsigned int*)0xFF3FF0C4 = 0x34E0A89;

if(!wait_0xFF3FF001_bit_6()) goto done;

*(unsigned int*)0xFF3FF0C0 = 0xD4398114;

if(!wait_0xFF3FF0CB()) goto done;

*(unsigned int*)0xFF3FF084 = 0x18;
*(unsigned int*)0xFF3FF0C4 = 0x410E0309;
*(unsigned int*)0xFF3FF004 = 0x40001;

if(!wait_0xFF3FF001_bit_6()) goto done;

set_key_to_0xFF3FF0C0((unsigned char*)key_0);

*(unsigned int*)0xFF3FF0C4 = 0x22450B19;

if(!wait_0xFF3FF001_bit_6()) goto done;

*(unsigned int*)0xFF3FF0C0 = 0x1D385210;
```

It looks like crypto functions have such logic:

1) Provide a seed for hash
2) Provide commands
3) Provide a data and keys
4) Provide hash to verify commands
5) Verify and execute commands

Another castle

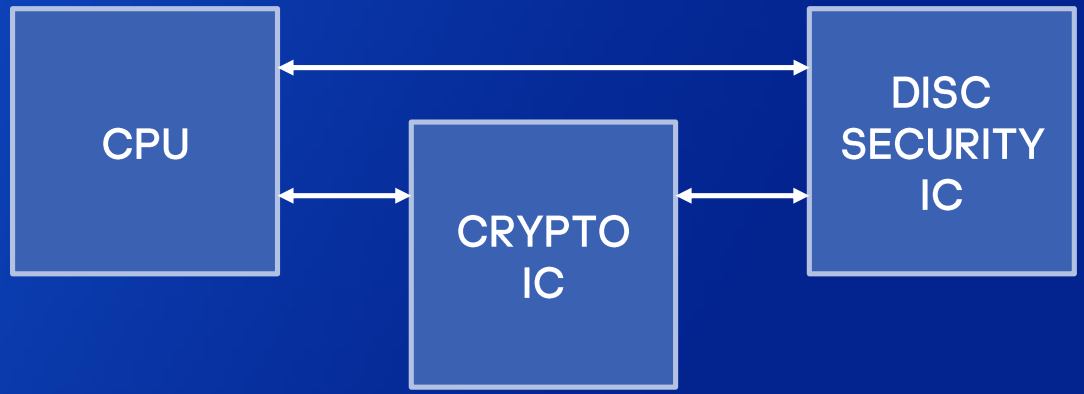
Played a little more with crypto processor but eventually lost interest

Breaking copy protection was never a goal

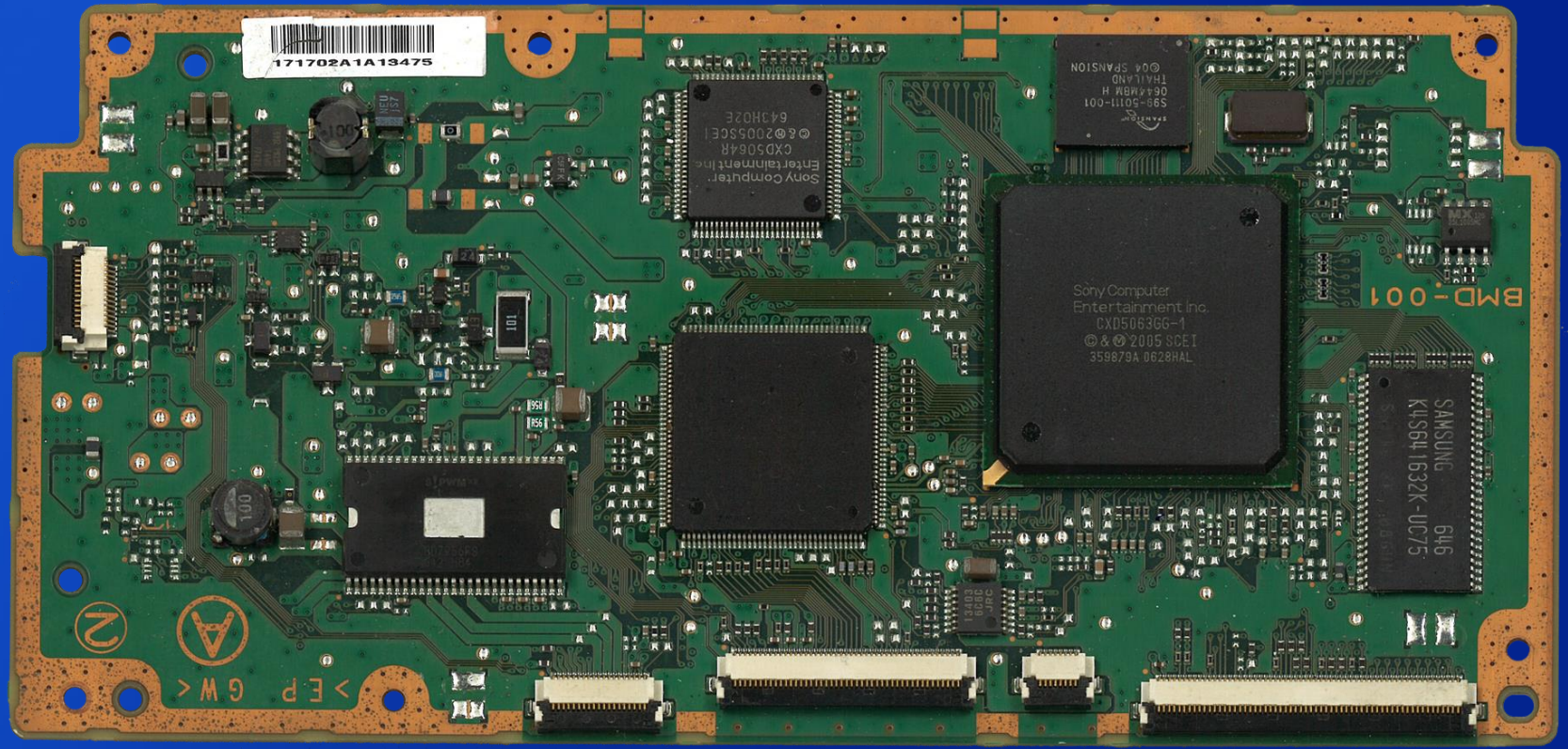
More reverse engineering revealed that crypto processor is likely exists only for crypto

Disc secrets are read and verified in another place, but its most likely done purely in hardware ☹

Understood that part with the help of firmware for the first PS3 retail drive – 302R

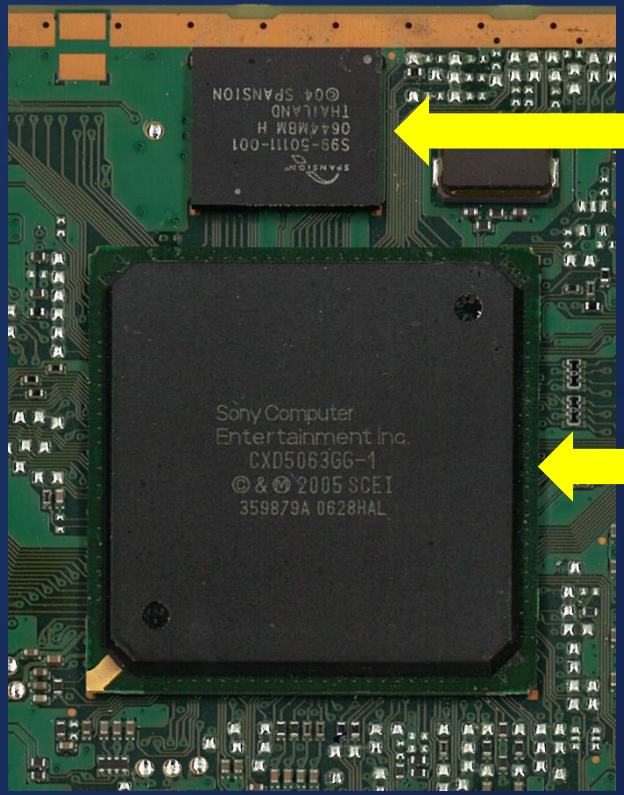


Hacking original Sony MCU



PCB of first PS3 retail drive

Hacking original Sony MCU (2)



1 MB NOR flash with firmware by Spansion

Main MCU is produced by Sony (Based on ARM)

Firmware is executed from external flash
Data is decrypted on fly
Of course encryption algorithm is based on XOR

Size of XOR stream is 0x10000 bytes

Find blocks of firmware filled with zeroes and partially recover XOR stream

Now it can be used to encrypt/decrypt pieces of firmware

Code is executed from external flash, but integrity of firmware is checked at boot

But still...

- We are able to observe all memory accesses from CPU to external flash with logic analyzer
- We are able to modify read address and data with FPGA
- We can write a small payload and encrypt it with recovered XOR stream
- We can modify memory accesses from CPU to execute our payload
- Use payload to read plaintext and leak it

✓ Code execution & firmware dump

Firmware is interesting

Unlike Renesas firmware it contains a lot of debug strings

Has a serial monitor with huge list of commands

List of commands:

ver, chgtsk, curtsk, dbgmode, setsqe, getsqe, sqrep, rtclk, sm, lt, lt_dsp, moni, spcboot, spcfwver, spcsaen, sacdwm, spcrmr, eeprom, spcrr, spcrw, osinfo, refflg, refmbx, refsem, refcyc, refalm, wallmsg, dsp, ddl, dpeek, ddump, dpoke ... and many more

Crypto processor

It also has a crypto processor but its very simple in comparison to Renesas:

- You just set keys, data, data size and command ID to special offsets in crypto MMIO region
- Initiate operation

It also has a firmware and its loaded from flash by bootrom

If you will try to read crypto MMIO region from flash you will get a garbage

You need to use a special functions in bootrom to access only specific offsets

Of course it can be bypassed with ROP, but all these functions also have integer overflows

So you can have a read and write access to the whole crypto MMIO region

Sony MCU and Renesas MCU are completely different systems

It means that all peripheral devices are different and accessed differently

But I found one peripheral device that is accessed exactly the same in both firmware's

The only difference in addresses:

- In Sony MCU registers of this peripheral device are mapped to special memory region
- In Renesas MCU these registers are accessed through Indigo DSP registers

Disc security IC (2)

I assume that this peripheral device is actually a disc security IC

It performs some interesting things

CRC of mystery disc data header is XOR'ed with "noki" and put into MMIO registers

If you modify it then crypto processor will not be able to return disc key

0A00h:	50	6C	61	79	53	74	61	74	69	6F	6E	34	00	00	00	00	PlayStation4....
0A10h:	43	55	53	41	2D	30	31	36	31	35	20	20	20	20	20	20	CUSA-01615
0A20h:	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
0A30h:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Mystery disc data header

\+ ? = Disc verify

Fun fact: "noki" is short for "enoki" – a mushroom that is well-known for its role in Japanese cuisine

Conclusions

I think Sony and partners did exceptional work, security model is really good and has proven itself

PlayStation Blu-ray drives existed since 2006 but no public hacks since then

But many have tried according to rumors

- 302R firmware hack was achieved for the first time by ps2dev guy back in 2006

Firmware can be hacked, so put all secrets to hardware

Crypto processor might be an interesting real world target if you are into glitching and SCA

September 17, 2008 – Cryptography Research Inc. (CRI) today announced an agreement with Renesas Technology Corp. regarding the use of CRI's patents to enhance the security of tamper-resistant smart card chips against Differential Power Analysis (DPA) and related attacks.

Props to everyone who also ever worked on hacking PlayStation Blu-ray drives independently

Thanks to you know who you are, this research would be not possible without you

Responsible disclosure

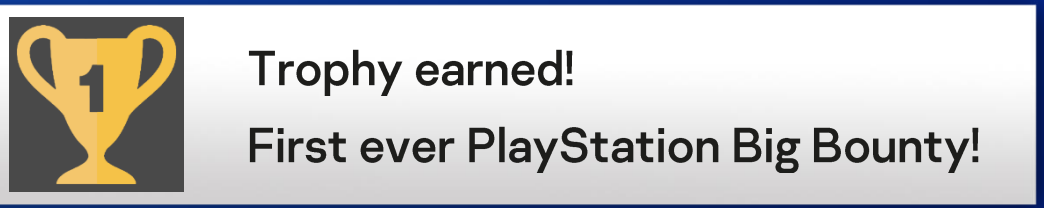
Nov 28, 2019 – Security team at Sony Interactive Entertainment reached out to me

Dec 03, 2019 – I provided information regarding vulnerabilities

Dec 10, 2019 – Security team invited me to join a recently launched Bug Bounty

Dec 12, 2019 – All vulnerabilities are triaged: 1 high and 2 medium severity bugs

Dec 19, 2019 – All vulnerabilities are fixed in system software version 7.02



Sony is about to announce something really awesome

I had a very pleasant experience with SIE security team

Slides will be uploaded here:

https://github.com/oct0xor/presentations

Thank you!

Boris Larin @oct0xor