



A BRIEF JOURNEY TO BAREMETAL HARDWARE HACKING





ΨHOAMI

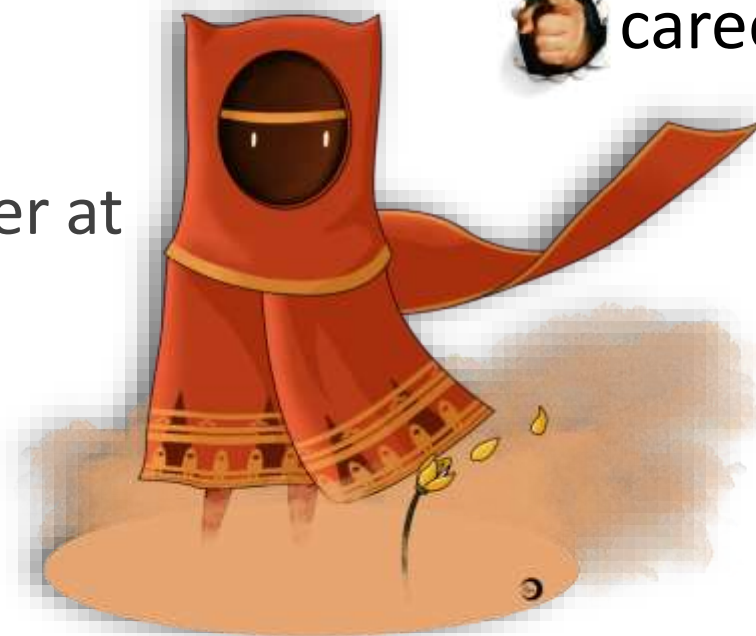


@jtsecES



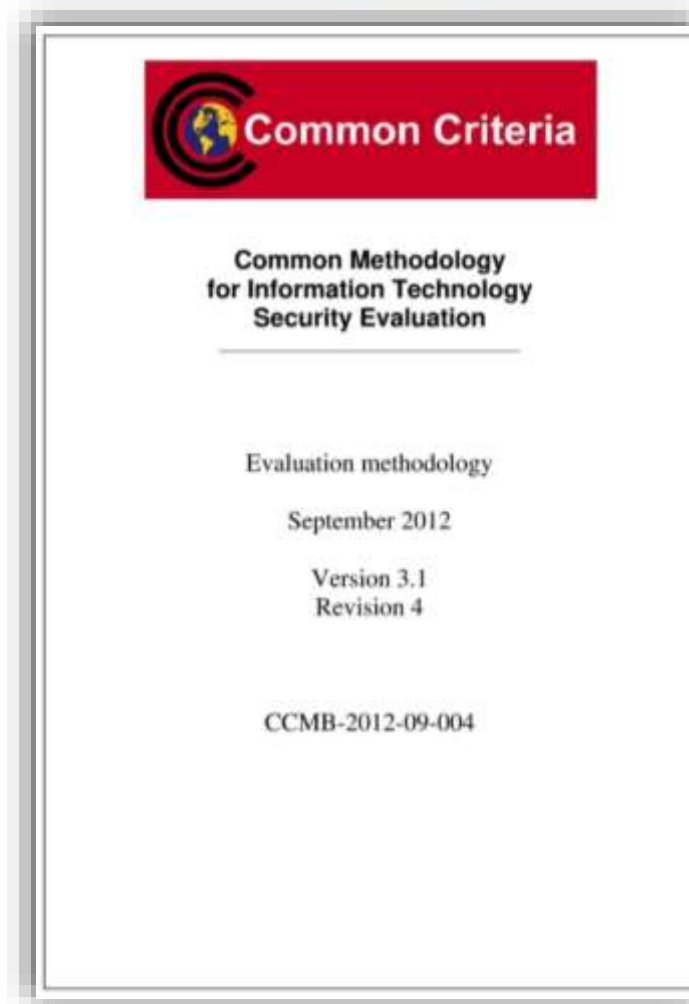
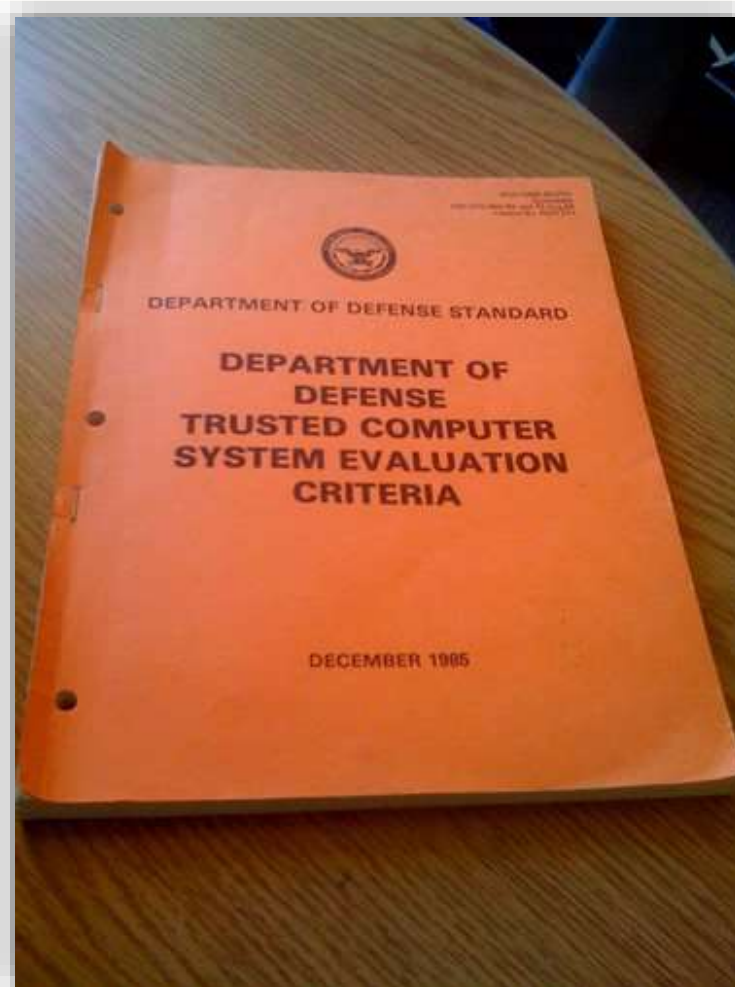
careers@jtsec.es

- Javier Tallón
- Co Founder and Chief Operations Officer at



- CyberSecurity Certification: Common Criteria, FIPS 140-2, ISO 27K1, ...
- Pentesters con sello





centro criptológico nacional



INSTITUTO NACIONAL DE CIBERSEGURIDAD





DISCLAIMER

- No podemos contaros cuál es el cacharro (ooooh!)
- Pero podemos poner fotos del interior (bieeeen!)
- Proyecto de pentesting (sin norma) para verificar la seguridad de su producto





BAREMETAL HARDWARE HACKING?

- ¿Qué entendemos por Baremetal?
- Trabajamos directamente sobre el hardware (metal)
- ¿Qué diferencias hay al hacer Ingeniería inversa?
 - No hay símbolos (NINGUNO)
 - No hay sistema de ficheros
 - Zero Knowledge





LAS ETAPAS DEL VIAJE

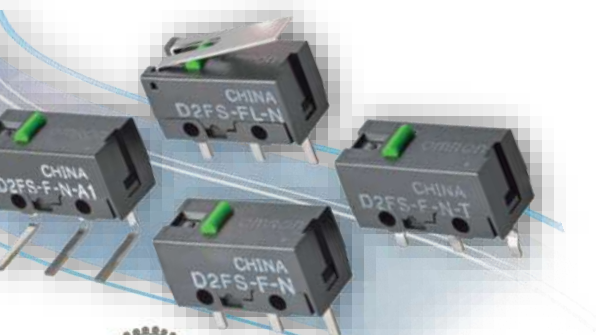
1. El Comienzo – El cacharro y su arquitectura
2. El Puente– Accediendo al código
3. El Desierto – Bypass de protecciones de lectura
4. El Descenso – Reversing ARM
5. Los Túneles – Debugging ARM
6. El Templo – Fuzz testing
7. La Montaña – Explotación
8. La Cumbre – Manteniendo el acceso





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Múltiples muestras permiten pruebas destructivas
- Primer paso: Abrir la caja
- Bypass Anti tamper Switch





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



- Fingerprint module
- LCD
- Contactless interface
- Contact interface
- USB interface
- 2x SAM cards





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



- No external flash or storage
- 2 x STM32 ARM ICs
 - APP: STM32F103RET6 <M3>
 - SEC: STM32F072 <M0>

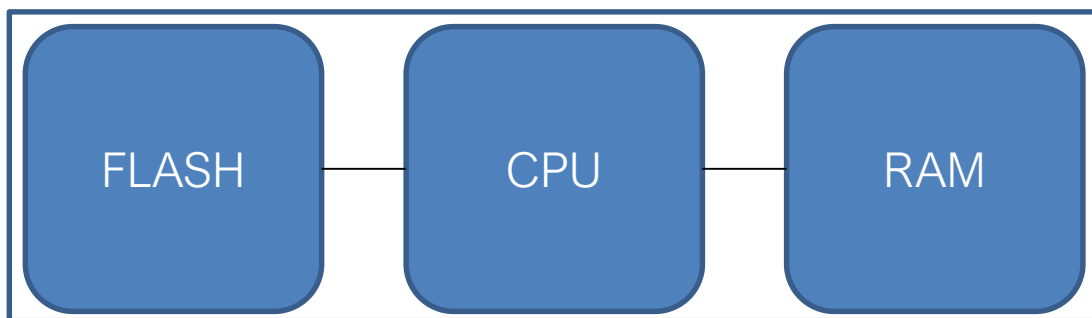
PDF
Data
Sheet



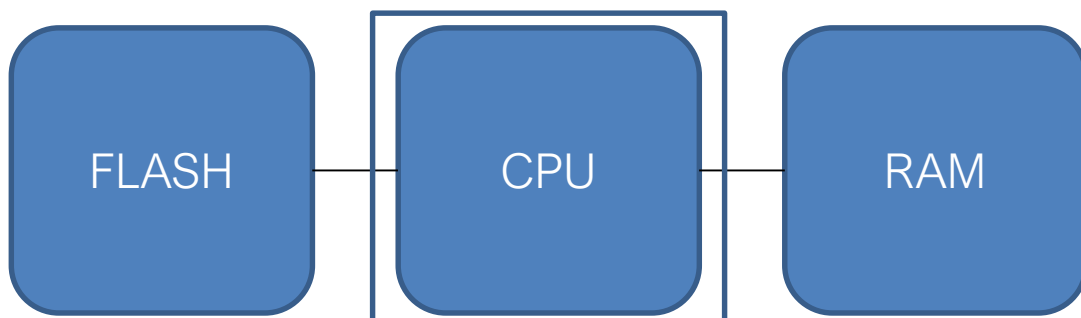


1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



- Microcontrolador
 - No suele haber S.O.
 - Poca capacidad
 - Capacidades de tiempo real



- Microprocesador
 - Flash menos protegida
 - Mayor capacidad





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



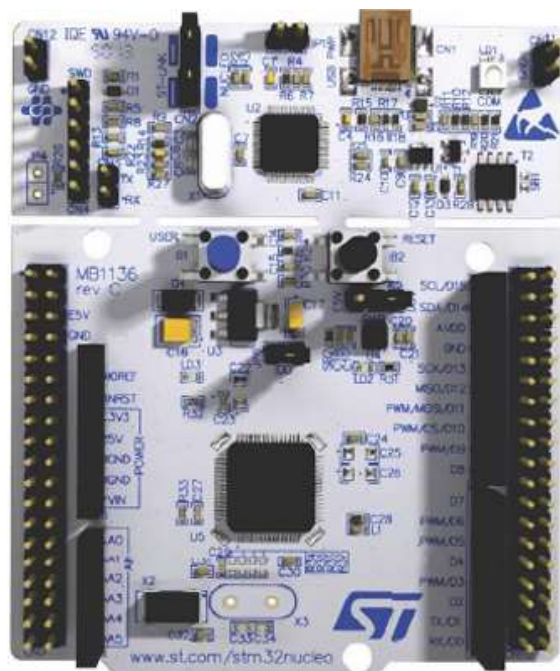
- STM32F103RET6 <M3>
 - Flash: 512KB
 - SRAM: 64 KB
- STM32F072RB <M0>
 - Flash: 128KB
 - SRAM: 16 KB





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



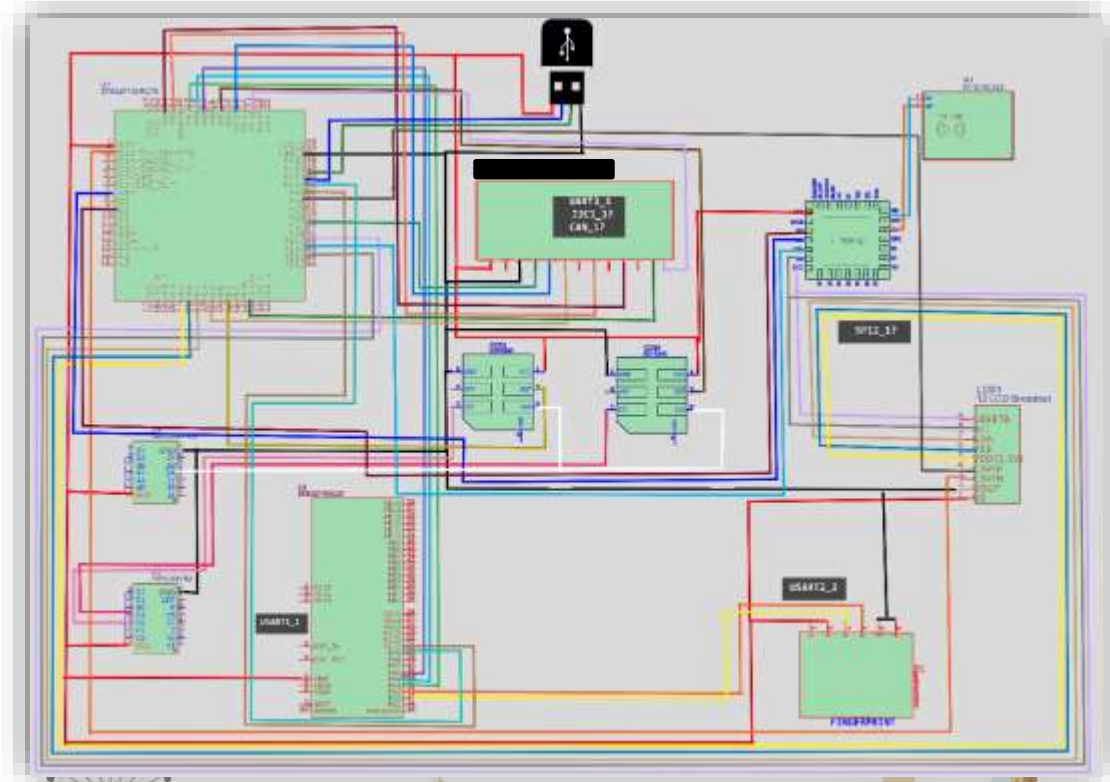
- Estudio detallado de la PCB





1. EL COMIENZO — EL CACHARRO Y SU ARQUITECTURA

- Fase de reconocimiento



- Identificamos los posibles paths de ataque
- Externos: USB, Fingerprint, Smartcard interfaces, SAM interfaces, LCD.
- Internos
 - JTAG/SWD
 - UART: IC1 – IC2, IC1 – Fingerprint IC1 – Smartcard, IC1 – SAM





2. EL PUENTE — ACCEDIENDO AL CODIGO

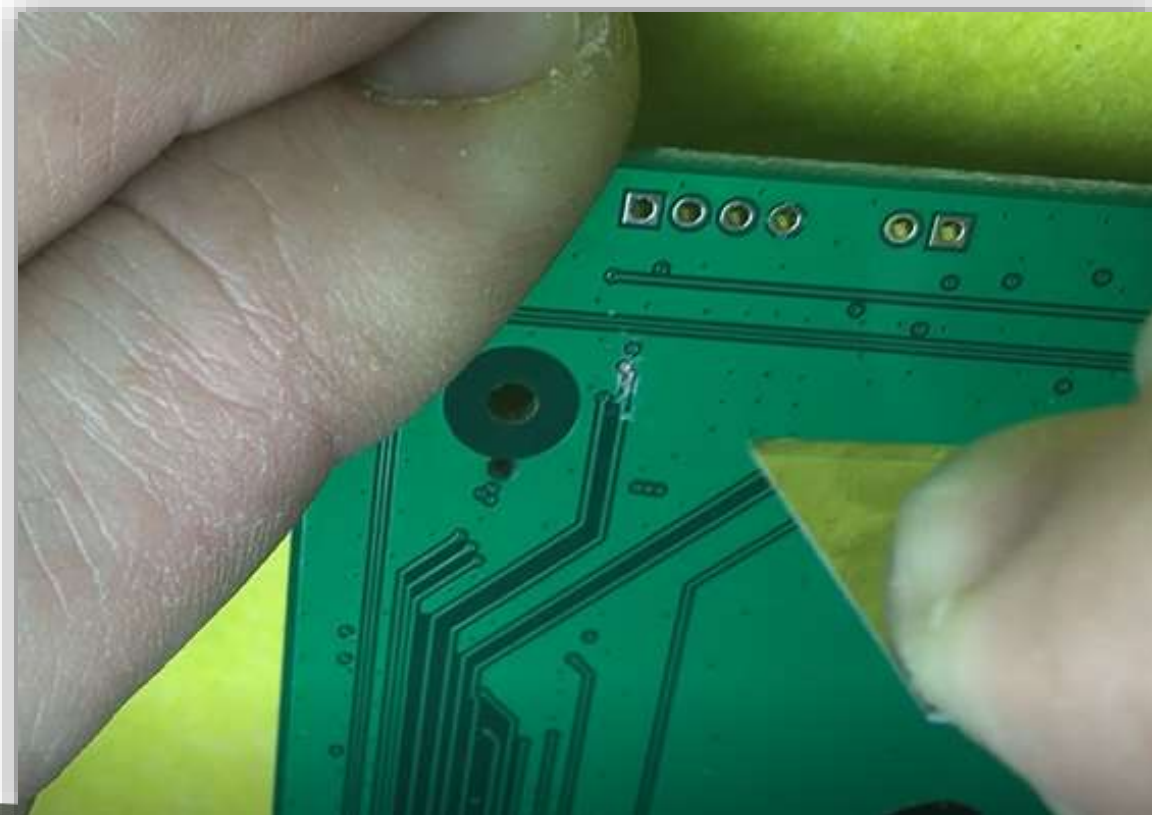
- El acceso a las interfaces no es tan fácil como a veces se pinta
 - Popular los PADS





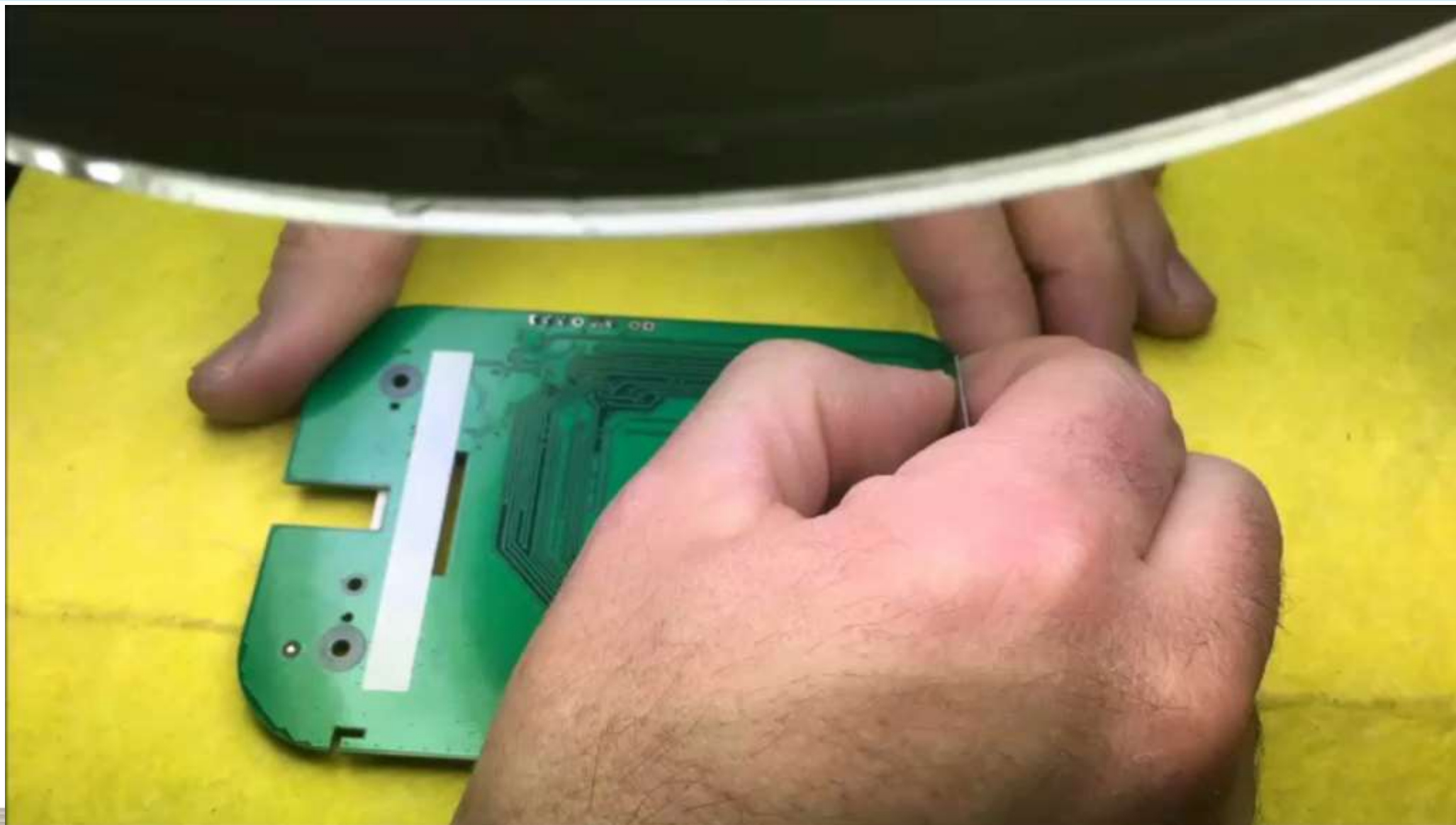
2. EL PUENTE — ACCEDIENDO AL CODIGO

- ¡Acceso directo a pista!





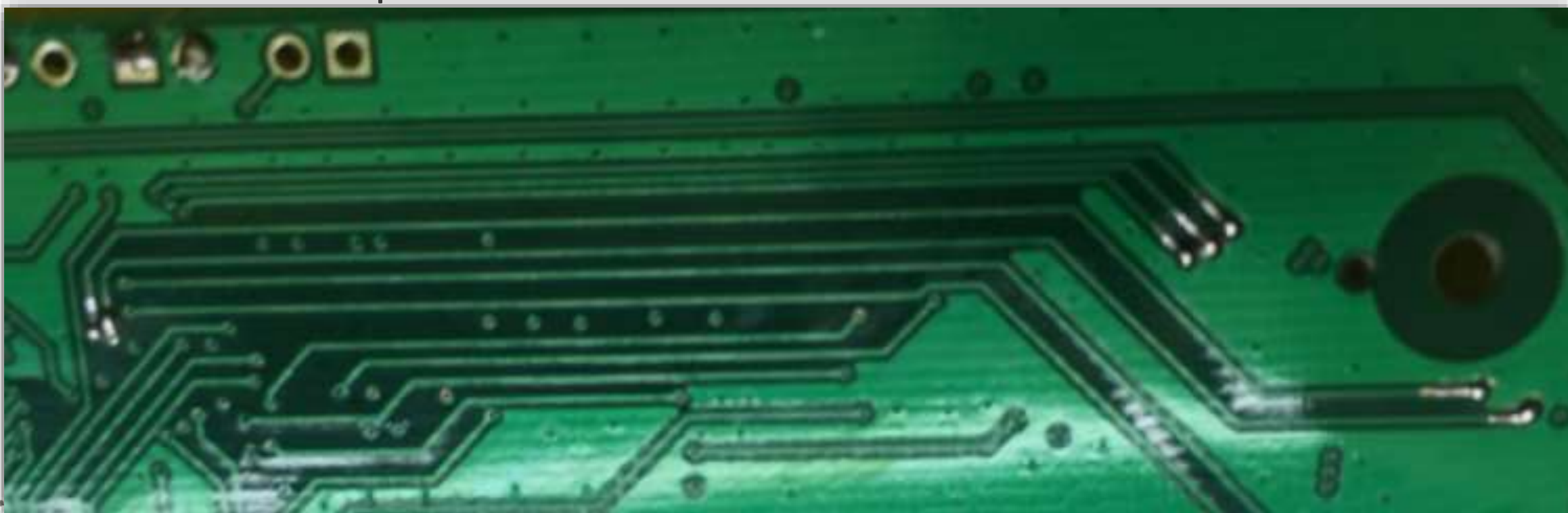
#CyberCamp18





2. EL PUENTE — ACCEDIENDO AL CODIGO

- ¡Acceso directo a pista!



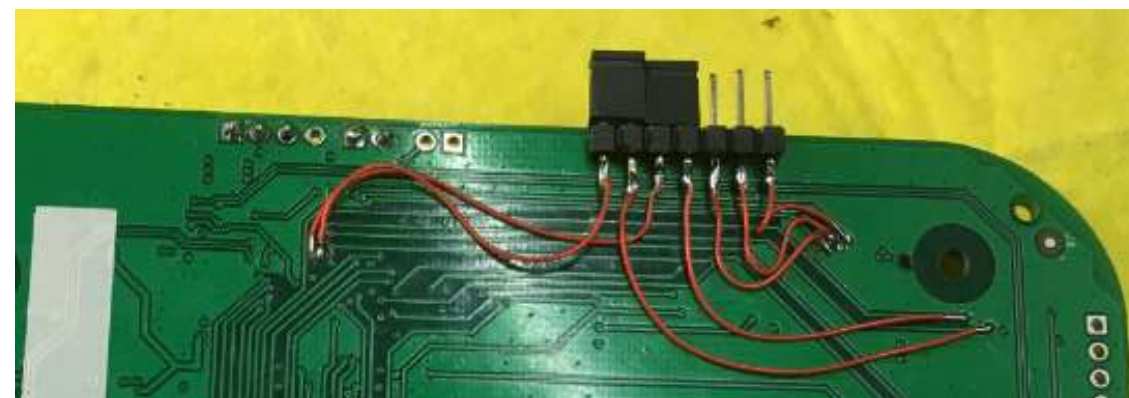
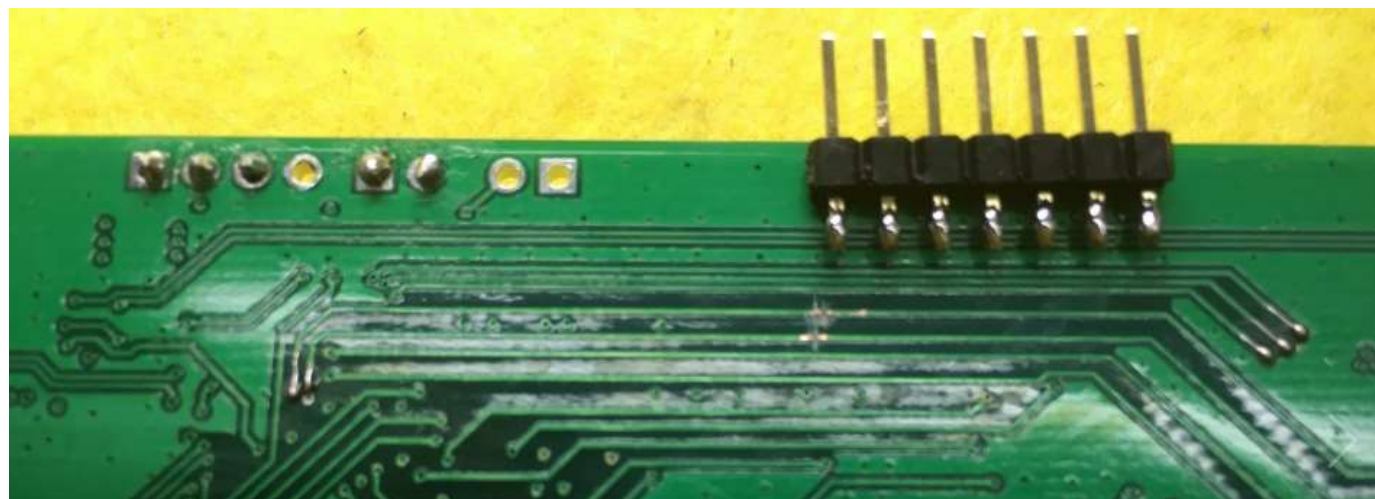


2. EL PUENTE — ACCEDIENDO AL CODIGO

- Acceso directo a pista!

UART

MAN IN THE BUS





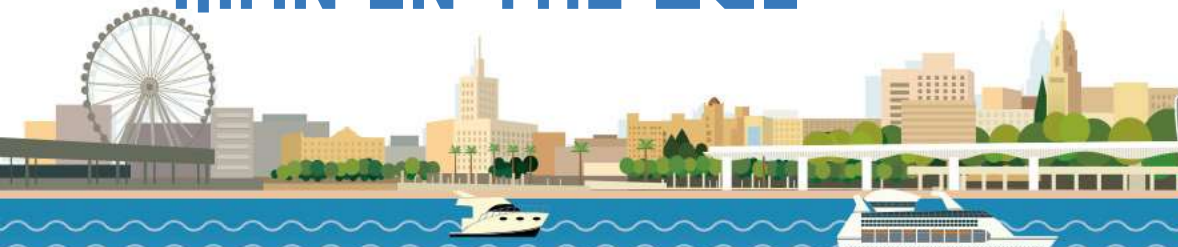
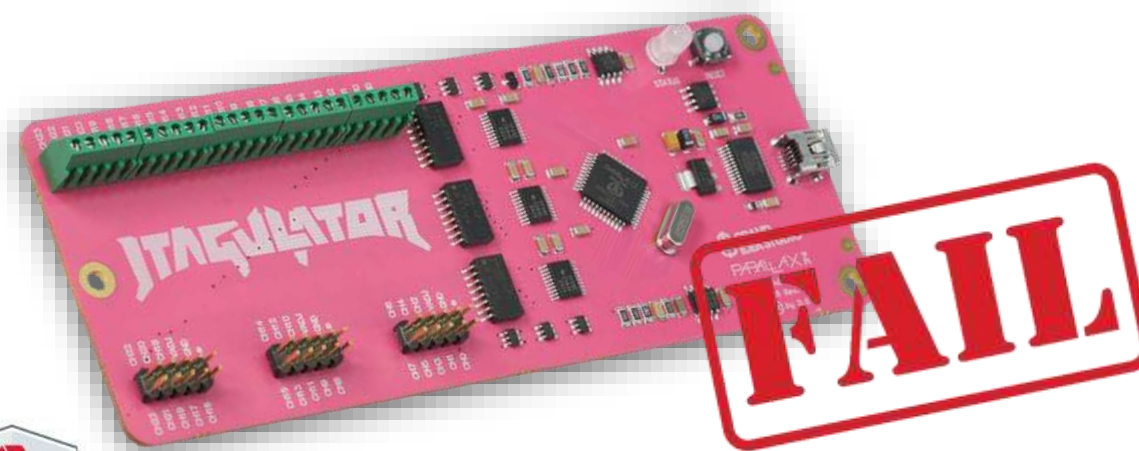


2. EL PUENTE — ACCEDIENDO AL CODIGO



MAN IN THE BUS

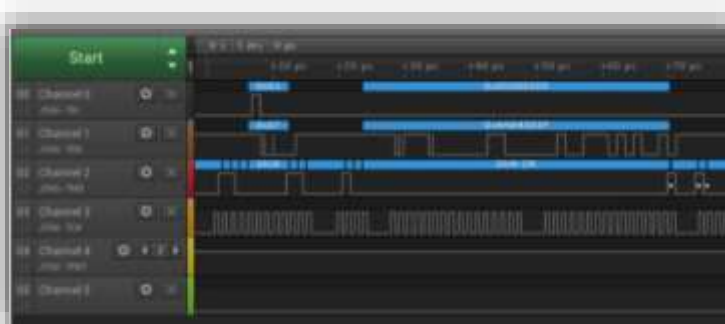
- Teóricamente podemos esnifar tráfico
- Hay que averiguar los parámetros de conexión





2. EL PUENTE — ACCEDIENDO AL CODIGO

- Teóricamente podemos esnifar tráfico
- Hay que averiguar los parámetros de conexión
- VELOCIDAD LOCA (approx. 921699 bps)



MAN IN THE BUS





2. EL PUENTE — ACCEDIENDO AL CODIGO

- Suplantando las comunicaciones y descifrando el protocolo



- Los paquetes empiezan por 02

```
00000000: 02 00 00 00 04 C0 06 00 00 C0 03 .....  
00000000: 02 00 00 00 09 00 00 00 30 56 30 2E 30 31 72 03 .....0V0.01r.  
00000000: 02 00 00 00 03 C0 0A 00 CB 03 .....  
00000000: 02 00 00 00 05 00 00 00 00 00 07 03 02 00 00 00 .....  
00000010: 0A 40 03 EE 00 FF 70 F2 00 01 FF 26 03 .@...p...&
```

- Terminan por 03

```
00000000: 02 00 00 00 05 80 00 EE 90 00 F9 03 .....  
00000000: 02 00 00 00 0B 40 03 EE 00 FF 70 04 00 02 00 02 .....  
00000010: 2F 03 .....  
00000000: 02 00 00 00 06 80 00 EE 00 90 00 FA 03 .....  
00000000: 02 00 00 00 0B 40 03 EE 00 FF 70 04 00 02 00 02 .....  
00000010: 2F 03 .....  
00000010: 2F 03 .....  
00000010: 2F 03 .....
```





2. EL PUENTE — ACCEDIENDO AL CODIGO

- Suplantando las comunicaciones y descifrando el protocolo



- Tiene un CRC

00000000: 02 00 00 00 04 C0 06 00 00 C0 03

Start byte

Number of info bytes

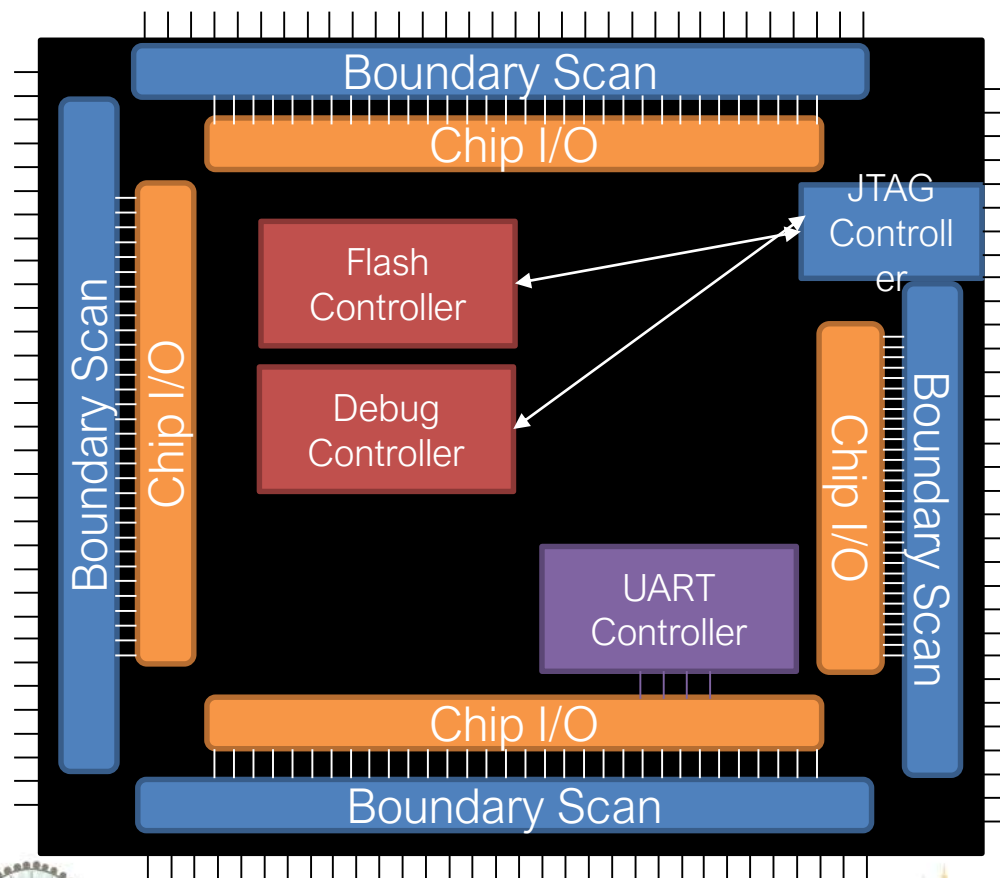
Info bytes

End
byte

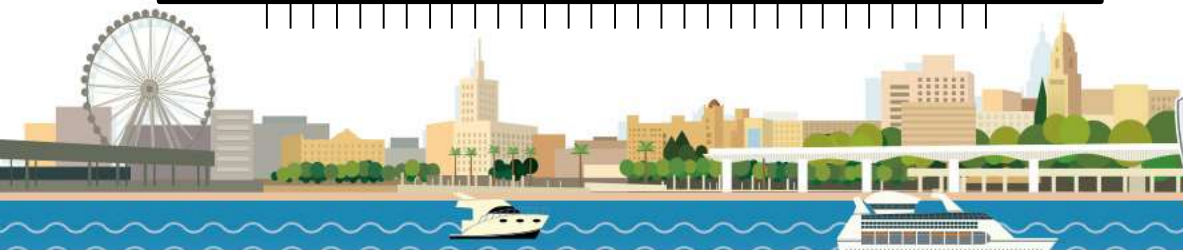




2. EL PUENTE — ACCEDIENDO AL CODIGO



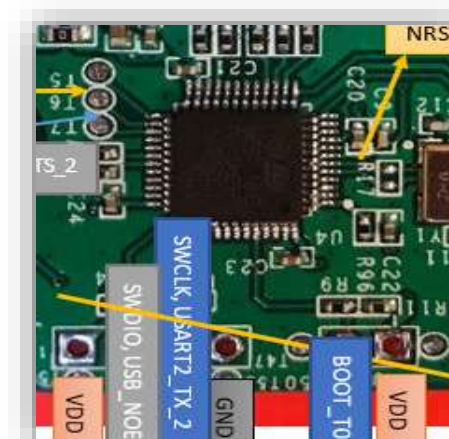
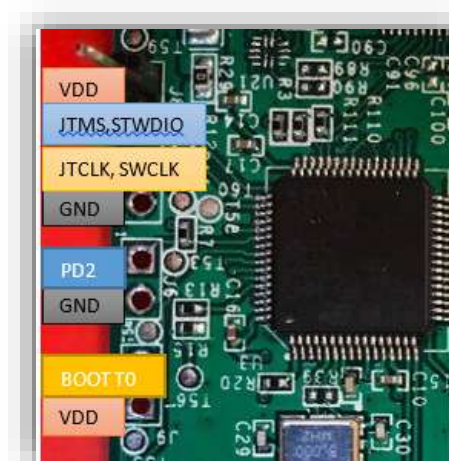
- Tres maneras de acceder al código
 - JTAG (Join Test Action Group)
 - Daisy chain
 - TDI, TDO, TMS, TCK, TRST
 - SWD
 - SWDIO, SWCLK
 - UART (Bootloader)
 - RX, TX, GND





2. EL PUENTE — ACCEDIENDO AL CODIGO

- JTAG no es estándar, pero en general es posible
 - Obtener identificación de los chips (IDCODE)
 - Controlar los pins (Boundary Scan)
 - Acceder a los registros de la CPU
 - Acceder a la RAM
 - Acceder a la Flash

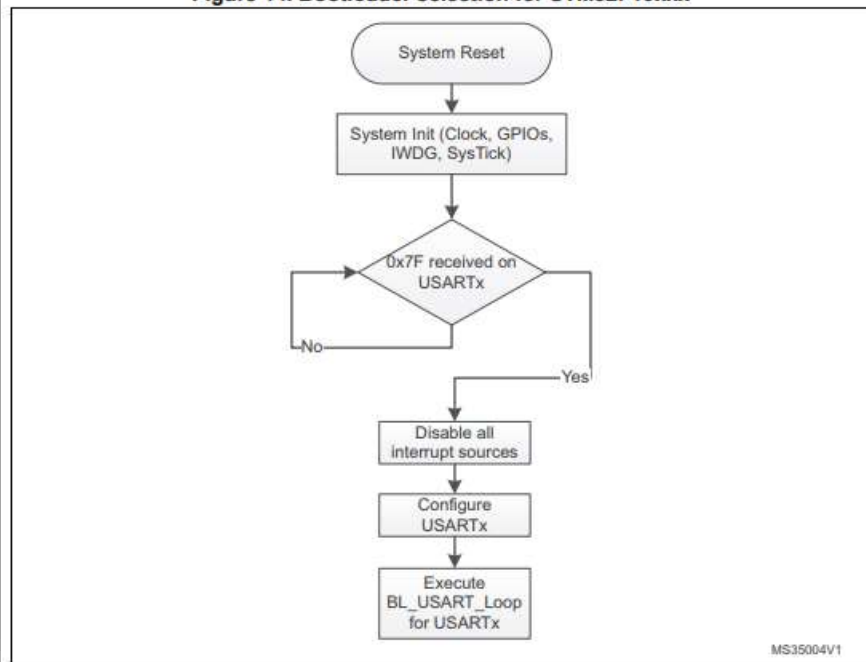




2. EL PUENTE — ACCEDIENDO AL CODIGO

- STM32F103RET6 (APP)

Figure 14. Bootloader selection for STM32F10xxx





2. EL PUENTE — ACCEDIENDO AL CODIGO

- STM32F103RET6 (APP)
 - Pudimos volcar el chip forzando a que el micro arrancase desde el bootloader y leyendo el puerto serie
 - Mantener en alta (3,3V) el pin BOOT0
 - Conectar al puerto serie (que normalmente conecta con el otro chip)
 - Forzar reset (no lo encontramos en la placa)



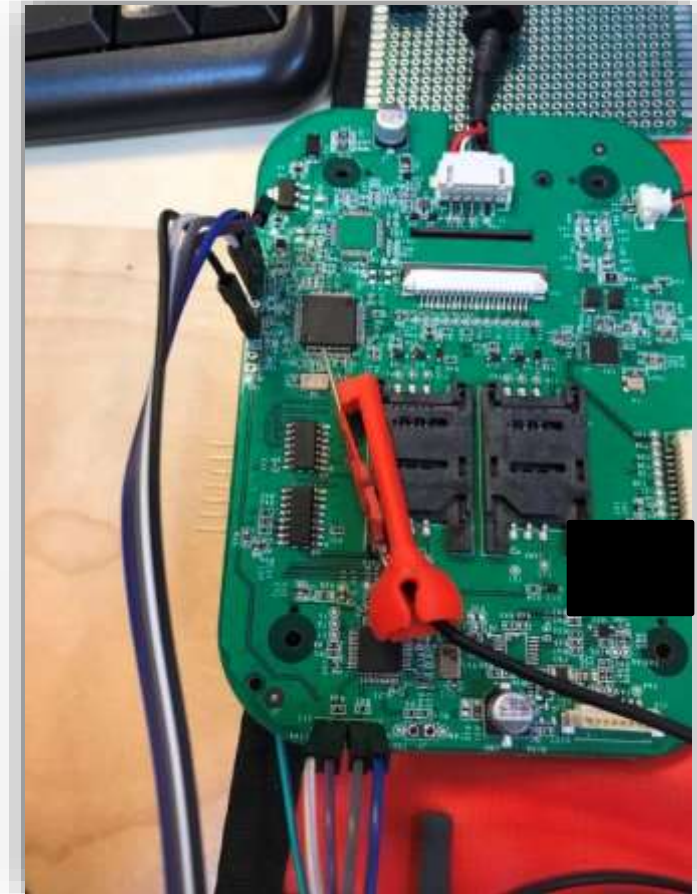


2. EL PUENTE — ACCEDIENDO AL CODIGO

- STM32F103RET6 (APP)
 - Pudimos volcar el chip forzando a que el micro arrancase desde el bootloader y leyendo el puerto serie
 - Mantener en alta (3,3V) el pin BOOT0
 - Conectar al puerto serie (que normalmente conecta con el otro chip)
 - Forzar reset (no lo encontramos en la placa)

Name	Start add...	End add...	Size	R	W
Page0	0x 80000...	0x 80003...	0x400 (1K)	Protected	Protected
Page1	0x 80004...	0x 80007...	0x400 (1K)	Protected	Protected
Page2	0x 80008...	0x 8000B...	0x400 (1K)	Protected	Protected
Page3	0x 8000C...	0x 8000F...	0x400 (1K)	Protected	Protected
Page4	0x 80010...	0x 80013...	0x400 (1K)	Protected	Protected
Page5	0x 80014...	0x 80017...	0x400 (1K)	Protected	Protected
Page6	0x 80018...	0x 8001B...	0x400 (1K)	Protected	Protected
Page7	0x 8001C...	0x 8001F...	0x400 (1K)	Protected	Protected
Page8	0x 80020...	0x 80023...	0x400 (1K)	Protected	Protected
Page9	0x 80024...	0x 80027...	0x400 (1K)	Protected	Protected
Page10	0x 80028...	0x 8002B...	0x400 (1K)	Protected	Protected
Page11	0x 8002C...	0x 8002F...	0x400 (1K)	Protected	Protected
Page12	0x 80030...	0x 80033...	0x400 (1K)	Protected	Protected
Page13	0x 80034...	0x 80037...	0x400 (1K)	Protected	Protected
Page14	0x 80038...	0x 8003B...	0x400 (1K)	Protected	Protected

Legend : Protected UnProtected





2. EL PUENTE — ACCEDIENDO AL CODIGO

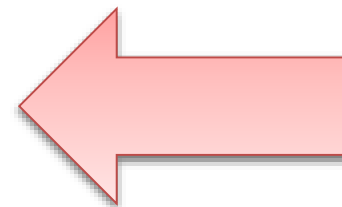
- STM32F072 (SEC)





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

- STM32F072 (SEC)
 - Tres niveles de seguridad para la protección de lectura (RDP)
 - 2 bytes: nRDP y RDP
 - $nRDP \neq \sim RDP$ (nRDP es el complemento bit a bit RDP)
- RDP Level 0: “no protection” (Default) Acceso complete de lectura/escritura
- RDP Level 1: “read protection” No hay acceso a la memoria flash
 - Permite salir del nivel, pero fuerza un borrado de todo
 - Pero permite el acceso a la SRAM
 - Y a los periféricos
- RDP Level 2: “no debug” SWD deshabilitado para siempre

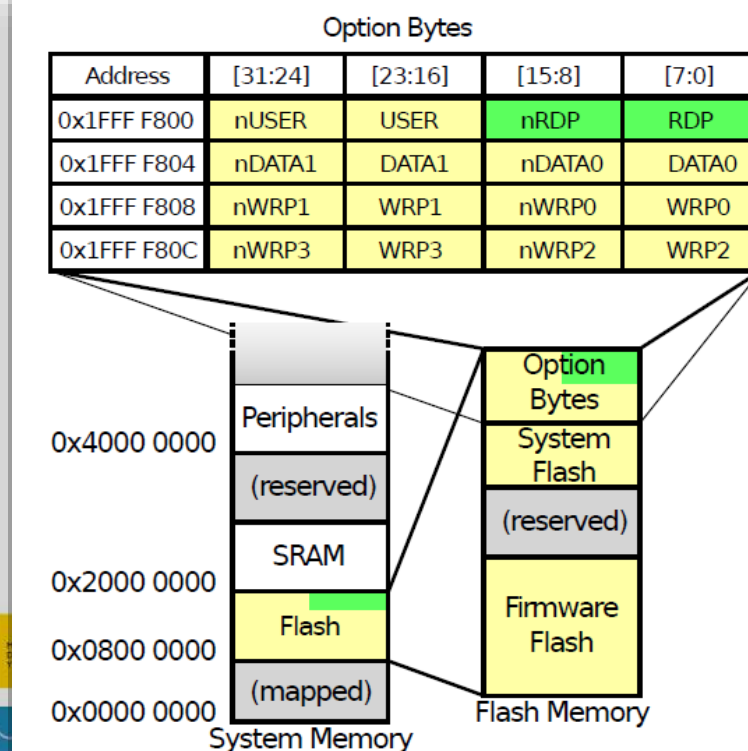




3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

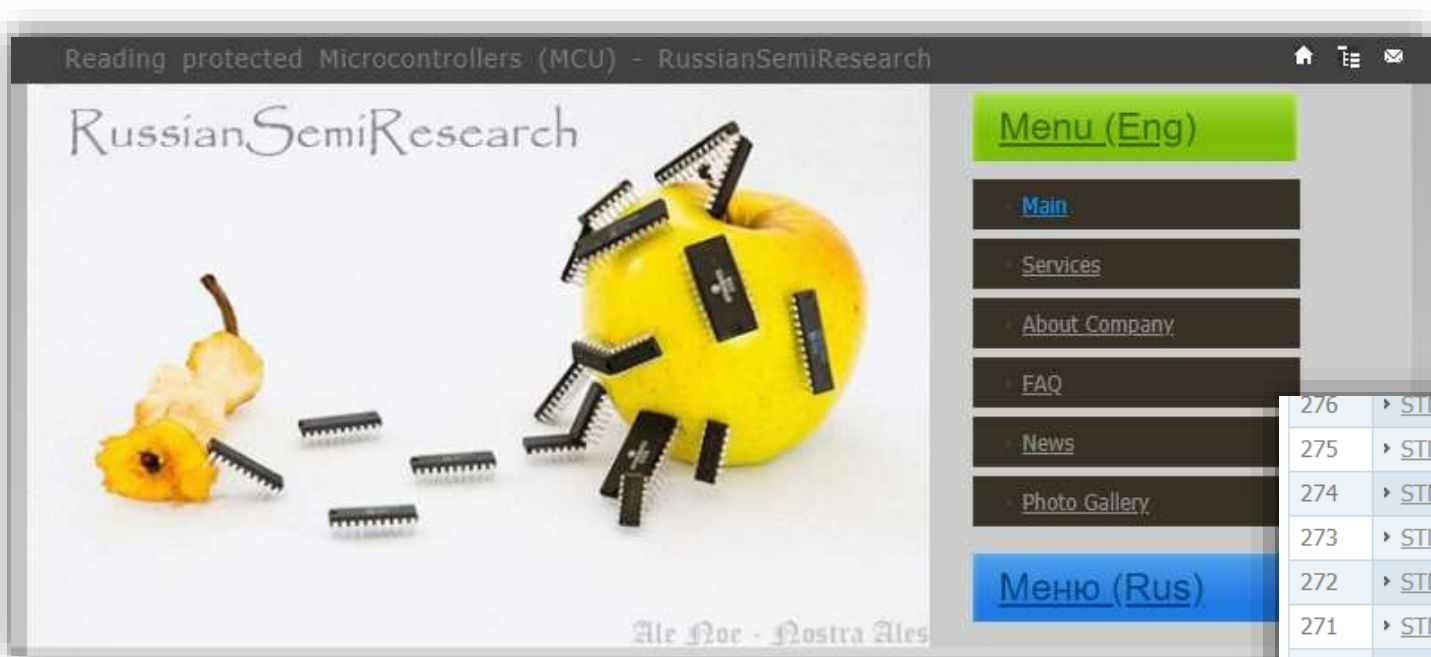
- STM32F072 (SEC)
 - RDP y nRDP: Guardados en la región “Option Bytes”
 - Memoria no volátil para la configuración del sistema
 - Parte de la memoria flash
 - Parte del mapa de memoria del sistema

nRDP	RDP	Protection
0x55	0xAA	RDP Level 0
Any other combination		RDP Level 1
0x33	0xCC	RDP Level 2





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA



276	▸ STMicroelectronics	STM32F103T6	\$2000
275	▸ STMicroelectronics	STM32F103T4	\$2000
274	▸ STMicroelectronics	STM32F103RG	\$2000
273	▸ STMicroelectronics	STM32F103RF	\$2000
272	▸ STMicroelectronics	STM32F103RE	\$2000
271	▸ STMicroelectronics	STM32F103RD	\$2000
270	▸ STMicroelectronics	STM32F103RC	\$2000
269	▸ STMicroelectronics	STM32F103R8	\$2000
268	▸ STMicroelectronics	STM32F103R6	\$2000
267	▸ STMicroelectronics	STM32F103R4	\$2000





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

- STM32F072 (SEC)
 - Tres ataques a la familia STM32F0
 - Cold Boot Stepping
 - Si podemos leer la SRAM y el código incluye una comprobación de integridad podemos inferir el contenido de la flash.
 - Security downgrade
 - Debug Interface exploit

Shedding too much Light on a Microcontroller's Firmware Protection

Johannes Obermaier
Fraunhofer Institute AISEC
johannes.obermaier@aisec.fraunhofer.de

Stefan Tatschner
Fraunhofer Institute AISEC
stefan.tatschner@aisec.fraunhofer.de

Abstract

Almost every microcontroller with integrated flash features firmware readout protection. This is a form of content protection which aims at securing intellectual property (IP) as well as cryptographic keys and algorithms from an adversary. One series of microcontrollers are the STM32 which have recently gained popularity and thus are increasingly under attack. However, no practical experience and information on the resilience of STM32 microcontrollers is publicly available. The paper presents the first investigation of the STM32 security concept, especially targeting the STM32F0 sub-series. Starting with a conceptual analysis, we discover three weaknesses and develop them to vulnerabilities by demonstrating corresponding Proofs-of-Concept. At first, we discover that a common security configuration provides low protection which can be exploited using our Cold-boot Stepping approach to extract critical data or even readout-protected firmware. Secondly, we reveal a design weakness in the security configuration storage which allows an attacker to downgrade the level of firmware protection, thereby enabling additional attacks. Thirdly, we discover and analyze a hardware flaw in the debug interface, attributed to a race condition, that allows us to directly extract read-protected firmware using an iterative approach. Each attack requires only low-priced equipment, thereby increasing the impact of each weakness and resulting in a severe threat altogether.

1 Introduction

Commercial grade microcontrollers are deployed in countless applications, ranging from industrial systems over automotive control units up to end-user devices. As their capabilities steadily increases, the complexity of their tasks rises and thus their firmware gets more sophisticated.

While previous devices were deployed in stand-alone applications, current systems may be part of large sensor networks or may interact with the Internet-of-Things. Thus, these systems contain valuable Intellectual Property (IP), such as sophisticated measurement or control algorithms. The devices may be license-locked and contain cryptographic material. Altogether, these devices are accompanied by large investments into software development.

At the same time, gaining access to these assets becomes more worthwhile for adversaries. Product piracy has emerged to a large threat, where competitors clone products and cause financial damage to the affected company [7]. As those attackers operate covertly without publishing their exploits, vulnerabilities are often surviving long. Nevertheless, professional researchers as well as hobbyists have also broken several systems in the past, often due to the underlying insufficient hardware security [16, 17]. Especially, Skorobogotov et al. have shown that the chosen security concepts of hardware manufacturers often do not cover all corner cases [13], have weaknesses [11], hidden functions, or even backdoors [12].

Many older microcontrollers were extensively tested for security and often exploited in the last few years. Therefore, the industry shows growing interest in more recent microcontrollers, including the ARM Cortex-M based STM32 series. The wide deployment of these devices finally raised interest into the provided security, mostly in terms of firmware protection.

There are no penetration testing results for STM32 publicly available. Thus, giving a statement regarding the protection of IP is impossible, despite it is often requested. Therefore we undertake a thorough security analysis of the STM32 series in which we answer the crucial question: Does the STM32 series provide a sufficiently strong security concept for firmware protection and, if not, how complex is the exploitation of weaknesses? We start with a high-level conceptual analysis of the security configuration and gradually dig deeper into the hardware implementation.





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

- STM32F072 (SEC)
 - Security Downgrade
 - 1 valor se mapea a CRP Level 2
 - 1 valor se mapea a CRP Level 0
 - El resto a CRP Level1

HEX		BIN				Flash Readout Protection
nRDP	RDP	nRDP	RDP	nRDP	RDP	
00	00	0000	0000	0000	0000	Level 2
00	01	0000	0000	0000	0001	
...	...	...	...	...	...	
33	CB	0011	0011	1100	1011	Level 1
33	CC	0011	0011	1100	1100	
33	CD	0011	0011	1100	1101	
...	...	...	...	...	...	Level 0
55	A9	0101	0101	1010	1001	
55	AA	0101	0101	1010	1010	
55	AB	0101	0101	1010	1011	
...	...	...	...	...	...	
FF	FE	1111	1111	1111	1110	
FF	FF	1111	1111	1111	1111	



2018



Shedding too much Light on a Microcontroller's Firmware Protection

Johannes Obermaier
Fraunhofer Institute AISEC
johannes.obermaier@aisec.fraunhofer.de

Stefan Tatschner
Fraunhofer Institute AISEC
stefan.tatschner@aisec.fraunhofer.de

Abstract

Almost every microcontroller with integrated flash features firmware readout protection. This is a form of content protection which aims at securing intellectual property (IP) as well as cryptographic keys and algorithms from an adversary. One series of microcontrollers are the STM32 which have recently gained popularity and thus are increasingly under attack. However, no practical experience and information on the resilience of STM32 microcontrollers is publicly available. The paper presents the first investigation of the STM32 security concept, especially targeting the STM32F0 sub-series. Starting with a conceptual analysis, we discover three weaknesses and develop them to vulnerabilities by demonstrating corresponding Proofs-of-Concept. At first, we discover that a common security configuration provides low protection which can be exploited using our Cold-boot Stepping approach to extract critical data or even readout-protected firmware. Secondly, we reveal a design weakness in the security configuration storage which allows an attacker to downgrade the level of firmware protection, thereby enabling additional attacks. Thirdly, we discover and analyze a hardware flaw in the debug interface, attributed to a race condition, that allows us to directly extract read-protected firmware using an iterative approach. Each attack requires only low-priced equipment, thereby increasing the impact of each weakness and resulting in a severe threat altogether.

While previous devices were deployed in stand-alone applications, current systems may be part of large sensor networks or may interact with the Internet-of-Things. Thus, these systems contain valuable Intellectual Property (IP), such as sophisticated measurement or control algorithms. The devices may be license-locked and contain cryptographic material. Altogether, these devices are accompanied by large investments into software development.

At the same time, gaining access to these assets becomes more worthwhile for adversaries. Product piracy has emerged to a large threat, where competitors clone products and cause financial damage to the affected company [7]. As those attackers operate covertly without publishing their exploits, vulnerabilities are often surviving long. Nevertheless, professional researchers as well as hobbyists have also broken several systems in the past, often due to the underlying insufficient hardware security [16, 17]. Especially, Skorobogatov et al. have shown that the chosen security concepts of hardware manufacturers often do not cover all corner cases [13], have weaknesses [11], hidden functions, or even backdoors [12].

Many older microcontrollers were extensively tested for security and often exploited in the last few years. Therefore, the industry shows growing interest in more recent microcontrollers, including the ARM Cortex-M based STM32 series. The wide deployment of these devices finally raised interest into the provided security, mostly in terms of firmware protection.

There are no penetration testing results for STM32 publicly available. Thus, giving a statement regarding the protection of IP is impossible, despite it is often requested. Therefore we undertake a thorough security analysis of the STM32 series in which we answer the crucial question: Does the STM32 series provide a sufficiently strong security concept for firmware protection and, if not, how complex is the exploitation of weaknesses? We start with a high-level conceptual analysis of the security configuration and gradually dig deeper into the hardware implementation.

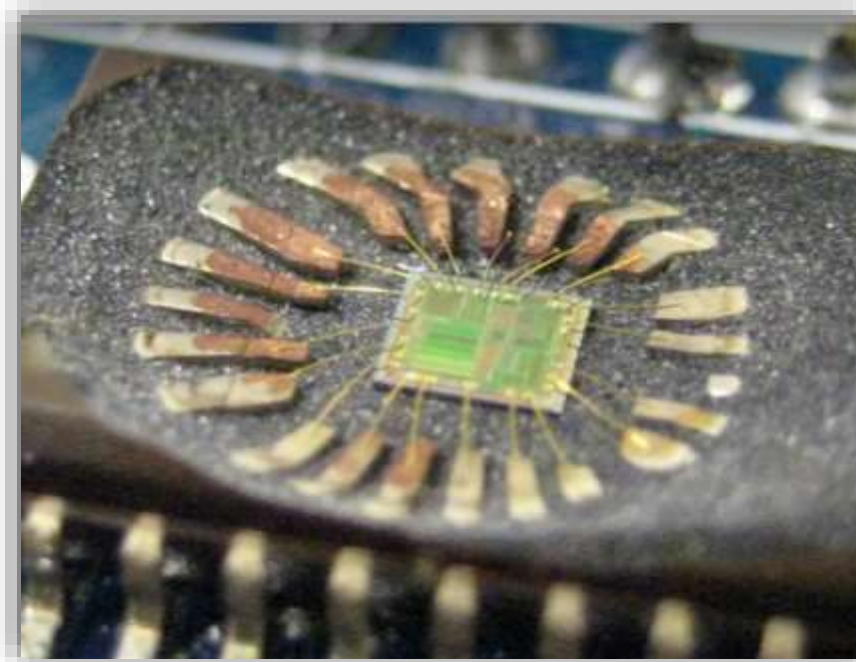
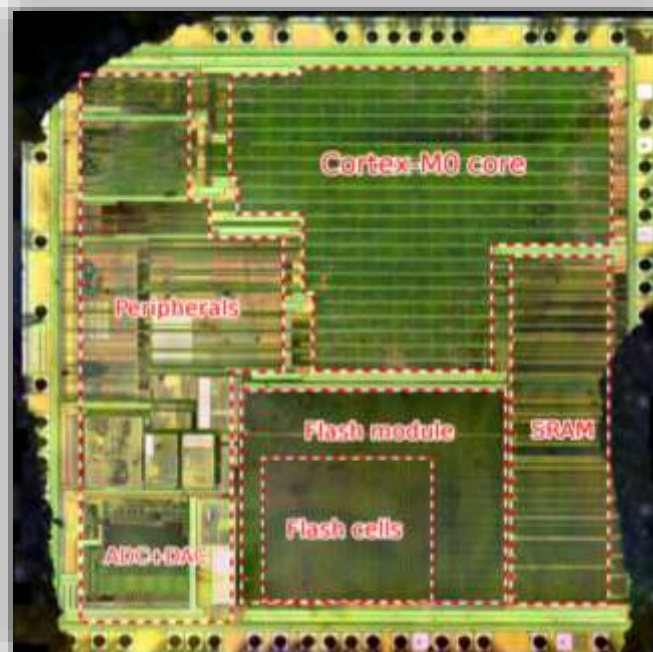
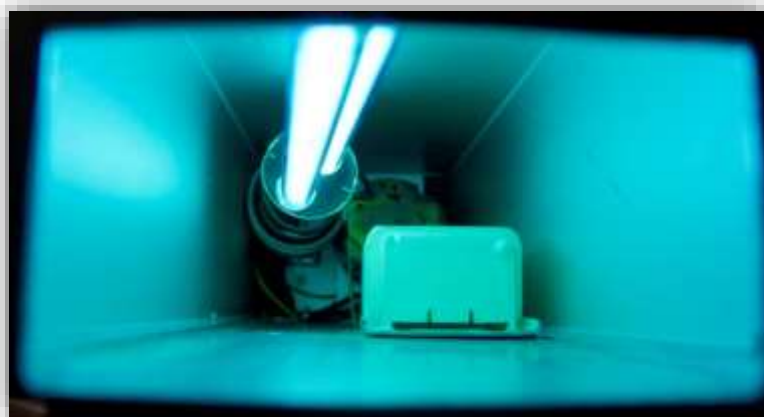
1 Introduction

Commercial grade microcontrollers are deployed in countless applications, ranging from industrial systems over automotive control units up to end-user devices. As their capabilities steadily increases, the complexity of their tasks rises and thus their firmware gets more sophisticated.



3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

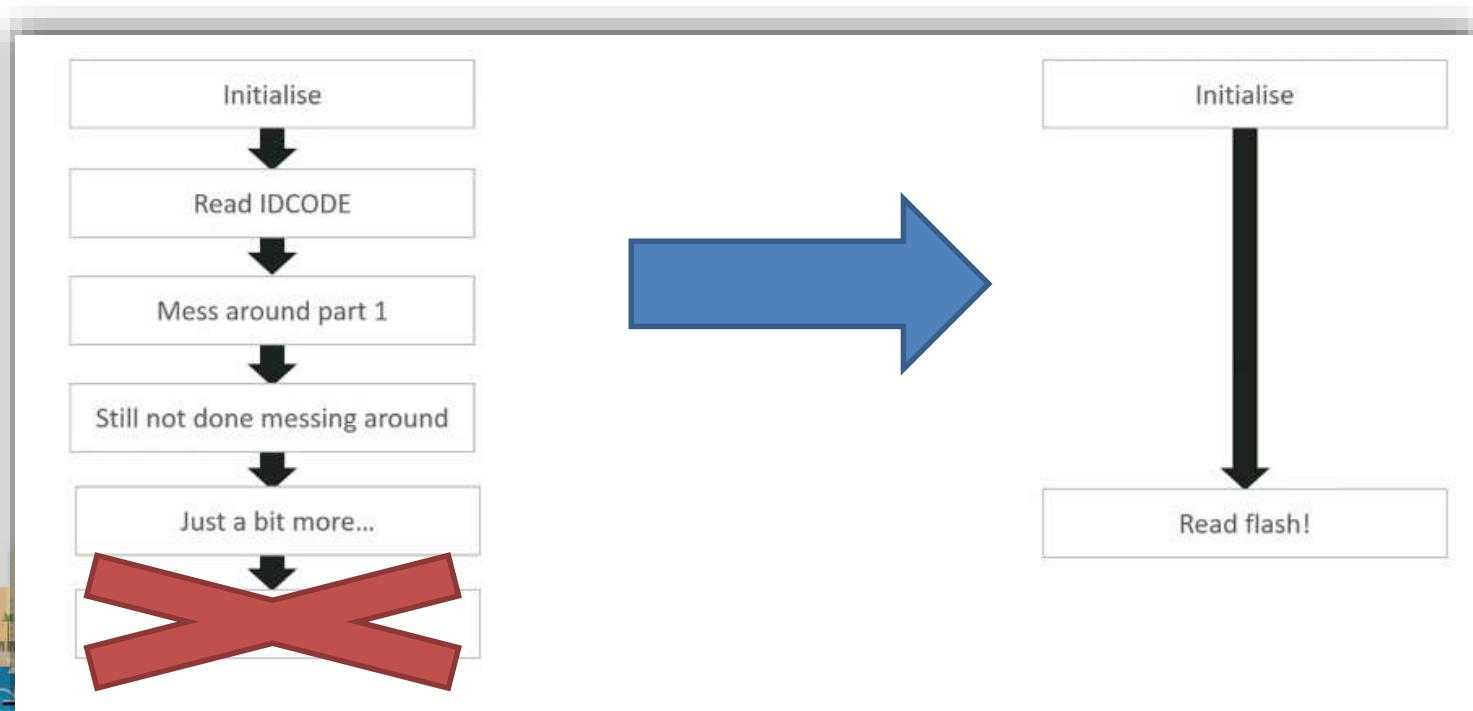
- STM32F072 (SEC)
 - Security Downgrade





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

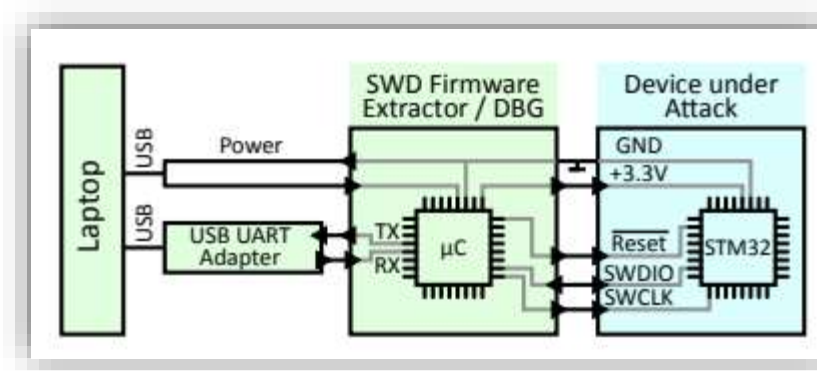
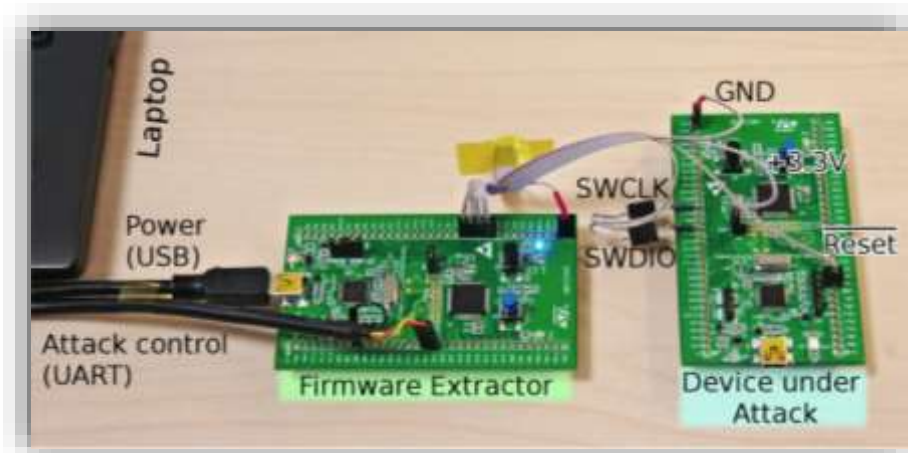
- STM32F072 (SEC)
 - Debug Interface exploit
 - Condición de carrera entre que se accede a la flash por SWD y que se deniega expresamente el permiso





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

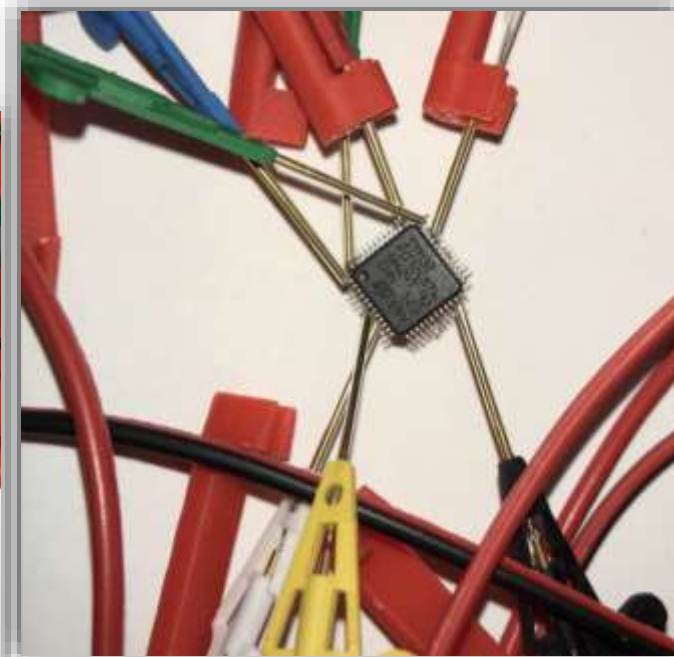
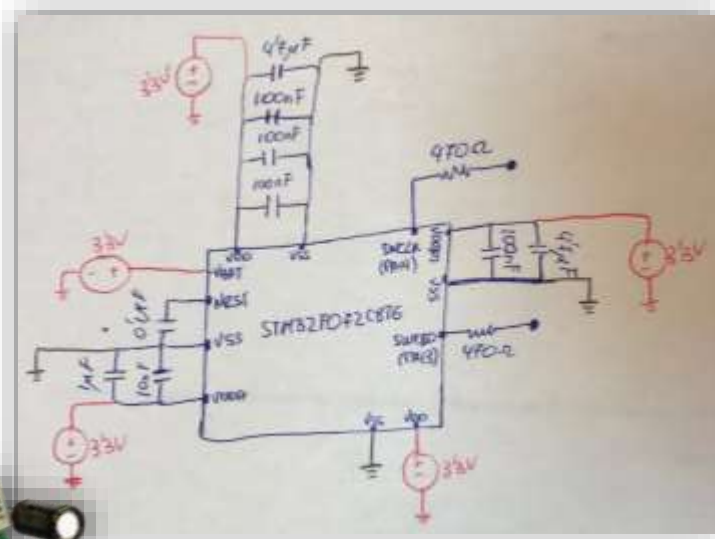
- STM32F072 (SEC)
 - Debug Interface exploit
 - Resetear sistema
 - Inicializar interfaz de debug
 - Configurar dirección de lectura
 - Leer flash
 - Si OK → address += 4





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

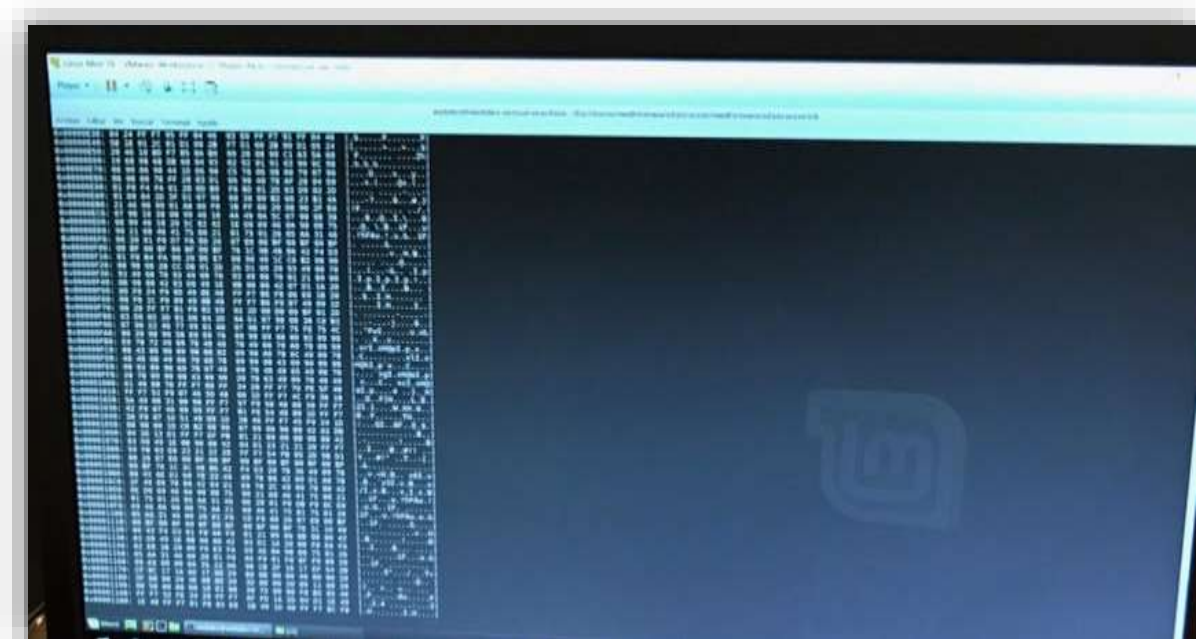
- STM32F072 (SEC)





3. EL DESIERTO — BYPASS DE PROTECCIONES DE LECTURA

- ¡Tenemos los dos binarios!





4. EL DESCENSO — REVERSING ARM

- ARM Assembly? No problem!
 - ARM es RISC, que es más simple que CISC
- **ARM ISA**
 - Modos CPU
 - ARM mode (4 B instructions) (word aligned)
 - Thumb mode (2 B instructions) (half word aligned)
 - Thumb-2 mode (2 or 4 B instructions)
 - El procesador sabe si está en Thumb mode porque el bit menos significativo del contador de programa vale 1
 - **Direcciones impares**





4. EL DESCENSO — REVERSING ARM

- **ARM ISA**

- Registros

- Todos son registros **R0-R15** generales y puede accederse directamente a ellos.
 - R15 se usa como EIP (**PC**).
 - R14 es el Link-Register (**LR**).
 - R13 se usa como ESP (**SP**)
 - R11 se usa como EBP (**FP**)

arm





4. EL DESCENSO — REVERSING ARM

- ARM ISA

- Saltos

- Branch (**B{cond} address**) → Salta a la dirección
 - Branch and optionally Exchange (**BX{cond} address**) → Salta a la dirección y cambia de modo si es necesario
 - Branch with Link (**BL{cond} address**) → Salta a la dirección y guarda la dirección de la siguiente instrucción en LR
 - Branch with Link and optionally Exchange (**BL{cond} address**) → Salta a la dirección, cambia de modo si es necesario y guarda la dirección de la siguiente instrucción en LR

- Acceso a memoria

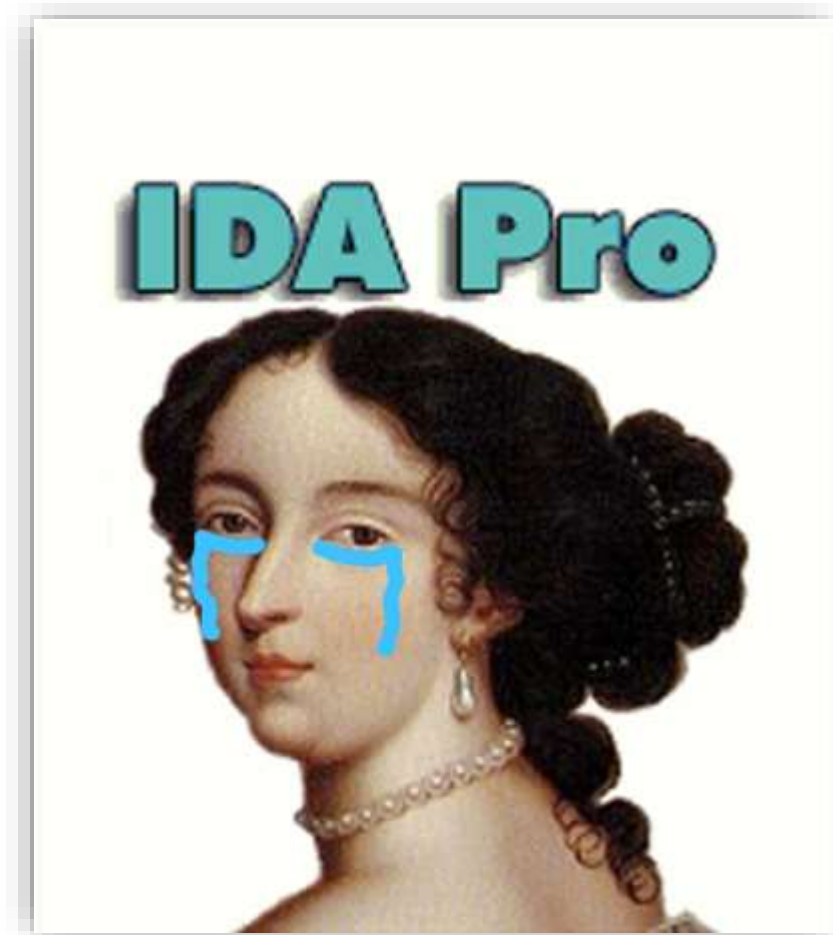
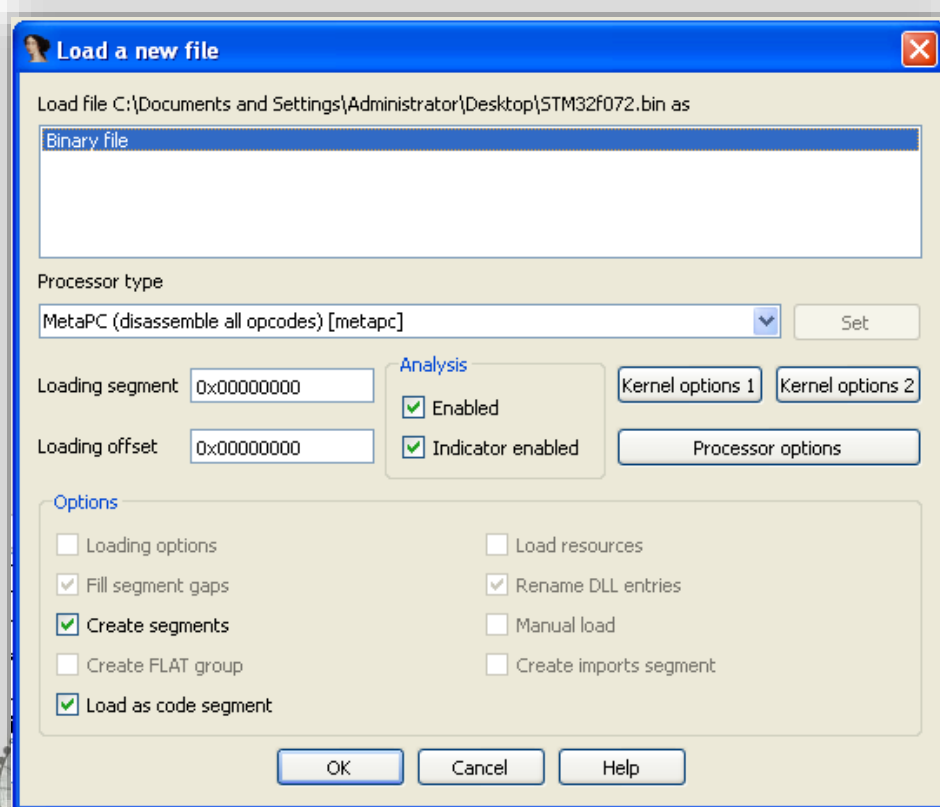
- **LDR Ra, [Rb]** → Copia en el registro Ra el contenido de la dirección apuntada por Rb
 - **STR Ra, [Rb]** → Copia en la dirección apuntada por Rb el contenido del registro Ra





4. EL DESCENSO — REVERSING ARM

- Where are my symbols?





4. EL DESCENSO — REVERSING ARM

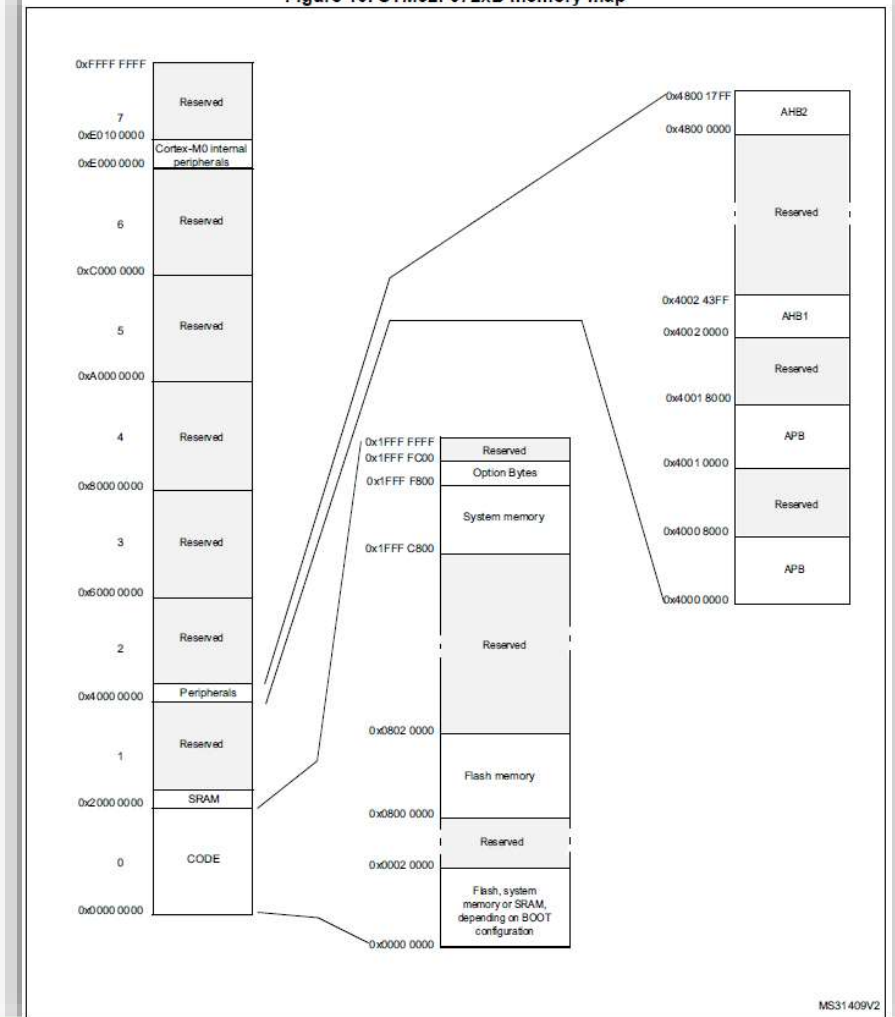
- Recovering symbols from datasheet

- First approach: Check datasheet!

Depending on the selected boot mode, main Flash memory, system memory or SRAM is accessible as follows:

- Boot from main Flash memory: the main Flash memory is aliased in the boot memory space (0x0000 0000), but still accessible from its original memory space (0x0800 0000). In other words, the Flash memory contents can be accessed starting from address 0x0000 0000 or 0x0800 0000.
- Boot from system memory: the system memory is aliased in the boot memory space (0x0000 0000), but still accessible from its original memory space (0x1FFF EC00 on STM32F03x and STM32F05x devices, 0x1FFF C400 on STM32F04x devices, 0x1FFF C800 on STM32F07x and 0x1FFF D800 on STM32F09x devices).
- Boot from the embedded SRAM: the SRAM is aliased in the boot memory space (0x0000 0000), but it is still accessible from its original memory space (0x2000 0000).

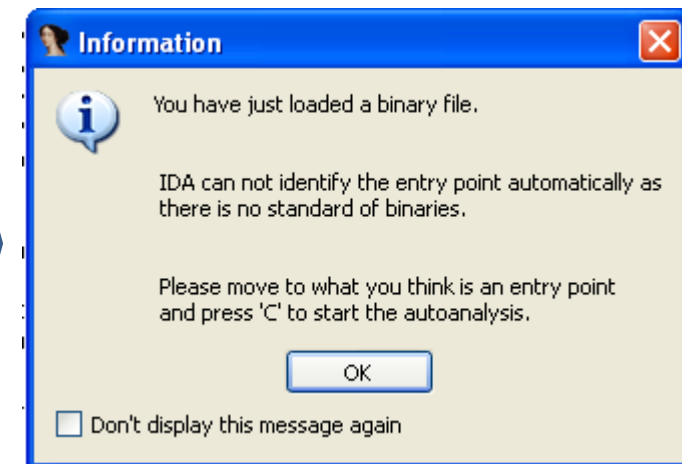
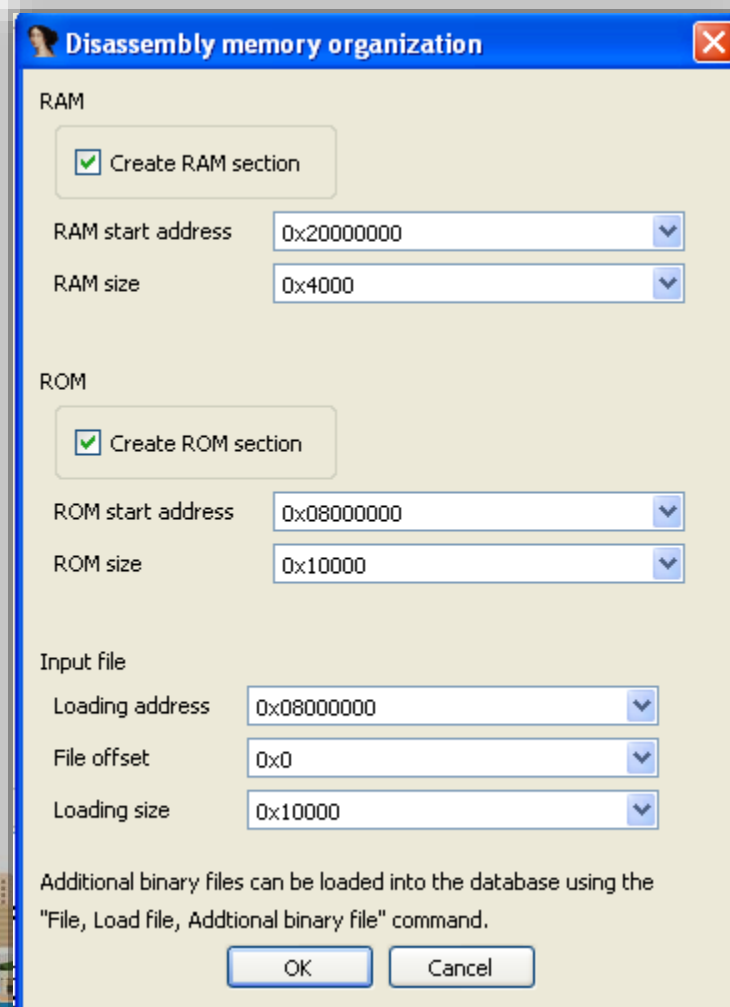
Figure 10. STM32F072xB memory map





4. EL DESCENSO — REVERSING ARM

- Recovering symbols



4. EL DESCENSO — REVERSING ARM

- Recovering symbols from datasheet
 - Los primeros bytes contienen la ISR, así que es un buen punto de partida
 - Reset apunta al código que se ejecutará en primer lugar

[illegible]

Interrupts and events RM0008

10.1.2 Interrupt and exception vectors

Table 61 and Table 63 are the vector tables for connectivity line and other STM32F10xxx devices, respectively.

Table 61. Vector table for connectivity line devices

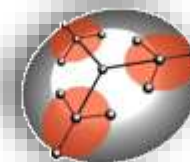
Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI	Non maskable Interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault	All class of fault	0x0000_000C
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV	Pendable request for system service	0x0000_0038
-	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	VWDG	Window Watchdog Interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection Interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper Interrupt	0x0000_0048
3	10	settable	RTC	RTC global Interrupt	0x0000_004C
4	11	settable	FLASH	Flash global Interrupt	0x0000_0050
5	12	settable	RCC	RCC global Interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 Interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 Interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 Interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 Interrupt	0x0000_0064
10	17	settable	EXTI4	EXTI Line4 Interrupt	0x0000_0068



4. EL DESCENSO — REVERSING ARM

- Recovering symbols with BinDiff
 - BinDiff es una herramienta de comparación para archivos binarios, que ayuda a los investigadores e ingenieros de vulnerabilidades a encontrar rápidamente diferencias y similitudes en el código desensamblado.
 - Con BinDiff puede identificar y aislar las correcciones de vulnerabilidades en los parches suministrados por los proveedores. También puede portar símbolos y comentarios entre desensamblajes de múltiples versiones del mismo binario o usar BinDiff para reunir pruebas de robo de código o violación de patente.
 - Necesita los archivos del IDB generados por IDA Pro.

zynamics



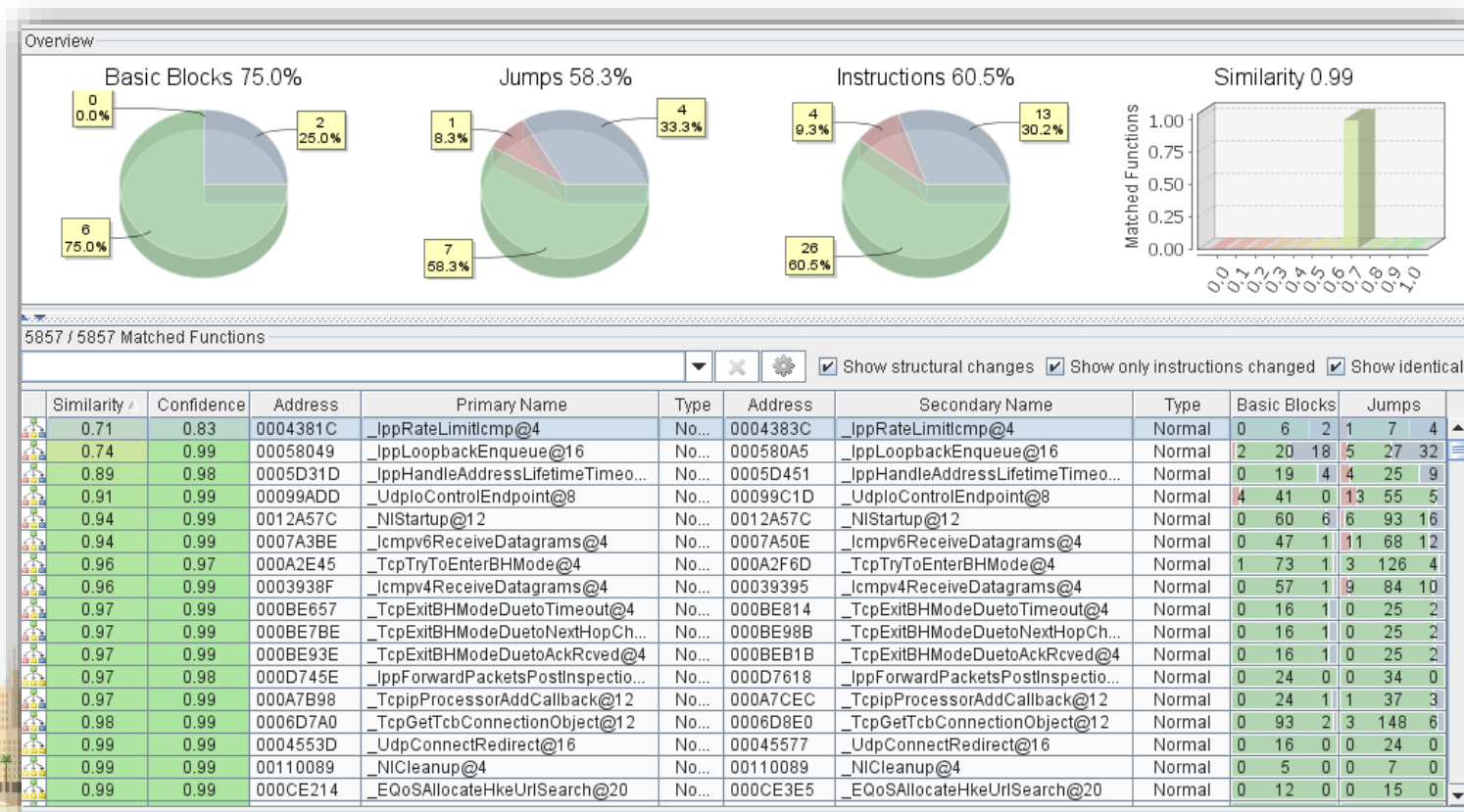
BinDiff





4. EL DESCENSO — REVERSING ARM

- Recovering symbols with BinDiff
 - BinDiff usa heurísticas para comprobar similitud entre funciones





4. EL DESCENSO — REVERSING ARM

- **Recovering symbols with BinDiff**
 - Podemos usarlo para:
 - Reusar trabajo entre el desensamblado de ambos MCUs
 - Adivinar el compilador usado y las funciones de libc.
 - Idea! Compilamos un programa básico con distintas toolchains y usamos BinDiff para encontrar similitudes
 - Supuesto general: Los desarrolladores son vagos y usarán lo que usa todo el mundo.
 - Googleamos “STM32 development environment”



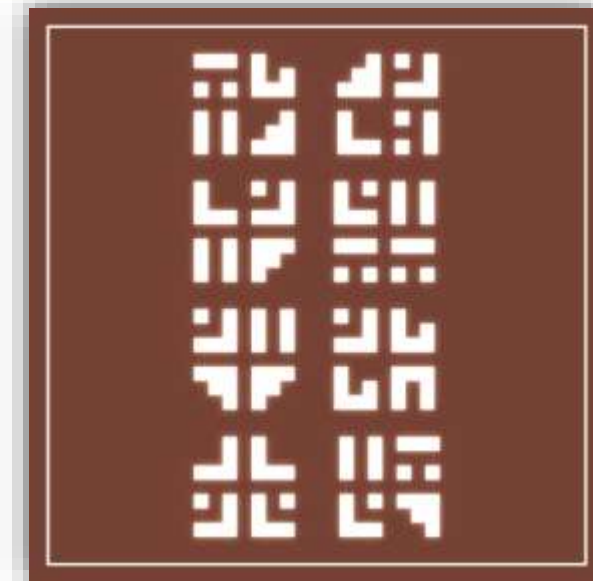
- **CONTIENE PROYECTOS Y APLICACIONES DE EJEMPLO!**





4. EL DESCENSO — REVERSING ARM

- Recovering symbols with BinDiff
 - Las toolchains / IDEs más importantes son:
 - IAR Embedded Workbench
 - Keil uVision
 - Atollic TrueStudio
 - Compilamos proyecto de ejemplo (con símbolos!), usamos BinDiff, si no funciona intentamos con más IDEs!



	Similarity ▾	Confidence ▾	Address ▾	Primary Name	Type ▾	Address	Secondary Name	Type	Basic Blocks			Jumps		
	1,00	0,95	08000728	sub_8000728	Normal	08000BAC	HAL_GetTick	Normal	0	1	0			
	0,97	0,99	080051B2	sub_80051B2	Normal	080001BA	__aeabi_memclr8	Normal	0	5	0	0	5	0
	0,96	0,98	08005180	sub_8005180	Normal	08000188	__aeabi_memcpy8	Normal	0	8	0	0	10	0
	0,85	0,90	080014C8	sub_80014C8	Normal	080039FA	USB_SetCurrentMode	Normal	0	7	0	0	8	0
	0,76	0,80	080051A4	sub_80051A4	Normal	08003A9E	__scatterload_zeroinit	Normal	0	4	0	0	4	0
	0,76	0,80	080026C0	sub_80026C0	Normal	080039DA	USB_ReadPacket	Normal	0	4	0	0	4	0





4. EL DESCENSO — REVERSING ARM

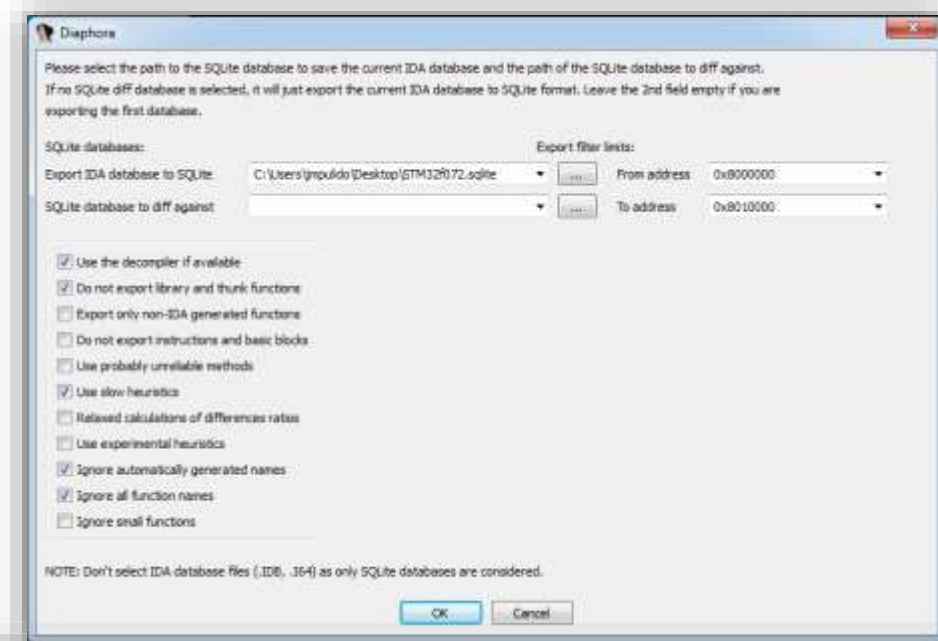
- **Recovering symbols with Diaphora**
 - Diaphora es un plugin para IDA Pro que tiene como objetivo ayudar en las tareas típicas de BinDiffing. Es similar a otros productos de la competencia y proyectos de código abierto como Zynamics BinDiff, DarunGrim o TurboDiff. Sin embargo, es capaz de realizar más acciones que cualquiera de los plugins o proyectos IDA anteriores.





4. EL DESCENSO — REVERSING ARM

- Recovering symbols with Diaphora
 - Script en Python que se llama desde IDA Pro
 - Usa bases de datos sqlite como almacenamiento intermedio.
 - Heurísticas como BinDiff pero usando el poder de Hex-Rays



Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00000	080001f4	__decompress1	080004ae	sub_80004AE	1.000	16	16	Same rare MD Index
00001	080001ac	__aeabi_memset8	080051a4	memset	1.000	4	4	Pseudo-code fuzzy AST hash
00002	08000188	__aeabi_memcpy8	08005180	memcpy	1.000	8	8	Strongly connected components
00003	080001ba	__aeabi_memclr8	080051b2	memclr	1.000	2	2	Callgraph match (caller of __aeabi_memset8/memset)

Line	Address	Name	Address 2	Name 2	Ratio	BBlocks 1	BBlocks 2	Description
00000	080002bc	CDC_Itf_Control	08001be8	sub_8001BE8	0.140	7	1	Same constants
00001	08002234	PCD_WriteEmptyTxFifo	0800743c	sub_800743C	0.220	12	13	Strongly connected components small-primes-product
00002	08003aac	free	08006dac	sub_8006DAC	0.330	15	13	Strongly connected components small-primes-product



4. EL DESCENSO — REVERSING ARM

- Recovering symbols with Diaphora

```
1 DWORD *__fastcall __aeabi_memcpy8(_DWORD *result, char *a2, unsigned int a3)
2 {
3     int v3; // r3@3
4     char v4; // t1@6
5     bool v5; // cf@7
6
7     if ( !(((unsigned int)result | (unsigned int)a2)
8     {
9         while ( a3 >= 4 )
10        {
11            v3 = *(_DWORD *)a2;
12            a2 += 4;
13            a3 -= 4;
14            *result = v3;
15            ++result;
16        }
17    }
18    while ( 1 )
19    {
20        v5 = a3-- >= 1;
21        if ( !v5 )
22            break;
23        v4 = *a2++;
24        *(_BYTE *)result = v4;
25        result = (_DWORD *)((char *)result + 1);
26    }
27    return result;
28}
```

```
1 DWORD *__fastcall memcpy(_DWORD *src, _BYTE *dst, unsigned int size)
2 {
3     int v3; // r3@3
4     char v4; // cf@7
5
6     if ( !((src | dst)
7     {
8         while ( size >= 4 )
9         {
10            v3 = *dst;
11            dst += 4;
12            size -= 4;
13            *src = v3;
14            ++src;
15        }
16    }
17    while ( 1 )
18    {
19        v4 = size-- >= 1;
20        if ( !v4 )
21            break;
22        *src = *dst;
23        src = (src + 1);
24        ++dst;
25    }
26    return src;
27}
```



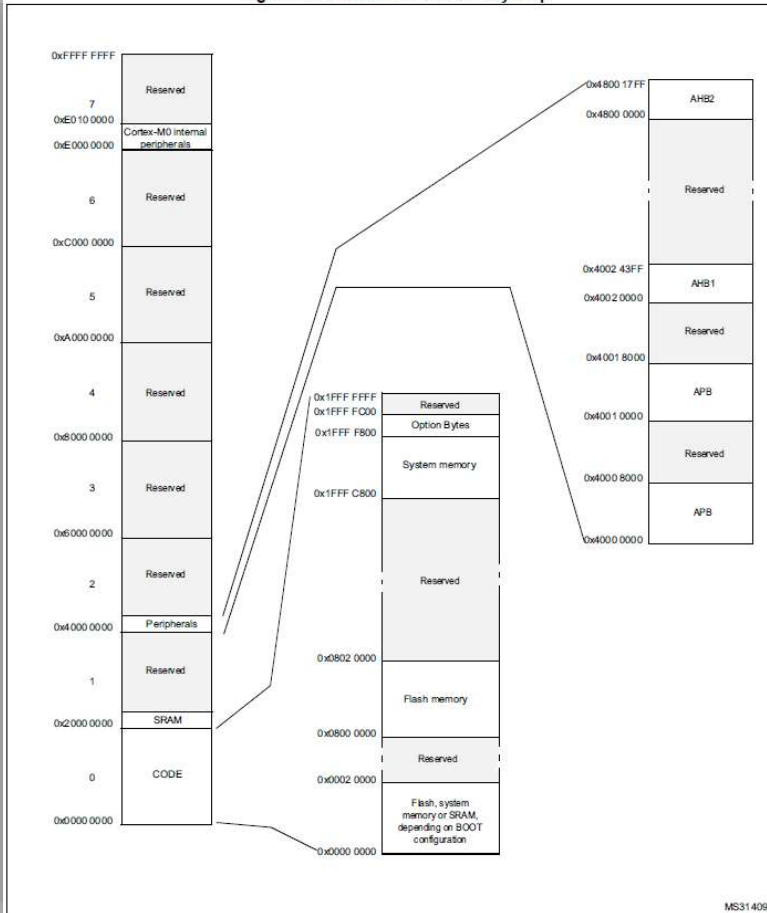


4. EL DESCENSO — REVERSING ARM

- IDAPython y el acceso a periféricos
- Check datasheet again!
- Los periféricos están mapeados en memoria

¿Cómo encontrar el acceso a los mismos en el desensamblado?

Figure 10. STM32F072xB memory map



Bus	Boundary address	Size	Peripheral
AHB2	0x4800 1800 - 0x5FFF FFFF	~384 MB	Reserved
	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE
	0x4800 0C00 - 0x4800 0FFF	1 KB	GIPOD
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB
AHB1	0x4002 4400 - 0x47FF FFFF	~128 MB	Reserved
	0x4002 4000 - 0x4002 43FF	1 KB	TSC
	0x4002 3400 - 0x4002 3FFF	3 KB	Reserved
	0x4002 3000 - 0x4002 33FF	1 KB	CRC
	0x4002 2400 - 0x4002 2FFF	3 KB	Reserved
	0x4002 2000 - 0x4002 23FF	1 KB	Flash memory interface
APB	0x4002 1400 - 0x4002 1FFF	3 KB	Reserved
	0x4002 1000 - 0x4002 13FF	1 KB	RCC
	0x4002 0400 - 0x4002 0FFF	3 KB	Reserved
	0x4002 0000 - 0x4002 03FF	1 KB	DMA
	0x4001 8000 - 0x4001 FFFF	32 KB	Reserved
	0x4001 5C00 - 0x4001 7FFF	9 KB	Reserved
APB	0x4001 5800 - 0x4001 5BFF	1 KB	DBGMCU
	0x4001 4C00 - 0x4001 57FF	3 KB	Reserved
	0x4001 4800 - 0x4001 4BFF	1 KB	TIM17
	0x4001 4400 - 0x4001 47FF	1 KB	TIM16
	0x4001 4000 - 0x4001 43FF	1 KB	TIM15
	0x4001 3C00 - 0x4001 3FFF	1 KB	Reserved
	0x4001 3800 - 0x4001 3BFF	1 KB	USART1
	0x4001 3400 - 0x4001 37FF	1 KB	Reserved
	0x4001 3000 - 0x4001 33FF	1 KB	SP1/I2S1
	0x4001 2C00 - 0x4001 2FFF	1 KB	TIM1
	0x4001 2800 - 0x4001 2BFF	1 KB	Reserved
	0x4001 2400 - 0x4001 27FF	1 KB	ADC
	0x4001 1800 - 0x4001 23FF	7 KB	Reserved
	0x4001 0400 - 0x4001 07FF	1 KB	EXTI
	0x4001 0000 - 0x4001 03FF	1 KB	SYSCFG + COMP
	0x4000 8000 - 0x4000 FFFF	32 KB	Reserved





4. EL DESCENSO — REVERSING ARM

- IDAPython y el acceso a periféricos
 - Sabiendo el aspecto que tendrá en el desensamblado la funcionalidad que estamos buscando podemos crear un script que busque ese patrón de ensamblador por todo el binario.
 - Para cada instrucción si el registro de la instrucción tiene la dirección de un periférico que nos interesa
Y
 - Es usado en un STR → Output
 - Es usado en un LDR → Input





4. EL DESCENSO — REVERSING ARM

- IDAPython y el acceso a periféricos
 - Sabiendo el aspecto que tendrá en el desensamblado la funcionalidad que estamos buscando podemos crear un script que busque ese patrón de ensamblador por todo el binario.

```
structuresSTM32F0 = {"FLASH_ACR": "0x40022000",  
    "FLASH_KEYR": "0x40022004",  
    "FLASH_OPTKEYR": "0x40022004",  
    "FLASH_SR": "0x4002200C",  
    "FLASH_CR": "0x40022010",  
    "FLASH_AR": "0x40022014",  
    "FLASH_OBR": "0x4002201C",  
    "FLASH_WRP": "0x40022020",  
    "RCC": "0x40021000",  
    "GPIOC": "0x48000800",  
    "GPIOA": "0x48000000",  
    "GPIOB": "0x48000400",  
    "GPIOD": "0x48000C00",  
    "GPIOE": "0x48001000",  
    "GPIOF": "0x48001400",  
    "CRC": "0x40023000",  
    "USART1": "0x40013800",  
    "USART2": "0x40004400",  
    "USART3": "0x40004800",  
    "USART4": "0x40004C00",  
    "I2C1": "0x40005400",  
    "I2C2": "0x40005800",  
    "SPI1": "0x40013000",  
    "SPI2": "0x40003800",  
    "USBCAN": "0x40006000",  
    "USBFS": "0x40005C00"}  
  
000529C 09 20 C0 06 MOVS R0, #0x40000000  
00052A0 01 F0 EE FE BL sub_0007000  
00052A4 E1 20 00 03 MOVS R0, #0x1000  
00052A8 00 20 STR R0, [SP, 0x28+var_28]  
00052AC 01 90 MOVS R0, #0  
00052B0 02 90 STR R0, [SP, 0x28+var_24]  
00052B4 03 90 STR R0, [SP, 0x28+var_20]  
00052B8 04 90 STR R0, [SP, 0x28+var_1C]  
00052BC 05 90 STR R0, [SP, 0x28+var_14]  
00052C0 0C 20 MOVS R0, #0xC  
00052C4 0A 90 STR R0, [SP, 0x28+var_10]  
00052C8 69 A6 MOV R0, SP  
00052CC 0C A6 LDR R0, =0x40013800  
00052D0 05 F0 FC FA BL sub_000A758 ; USART1 (reading from regist  
00052D4 1B 21 MOVS R1, #0x1B
```

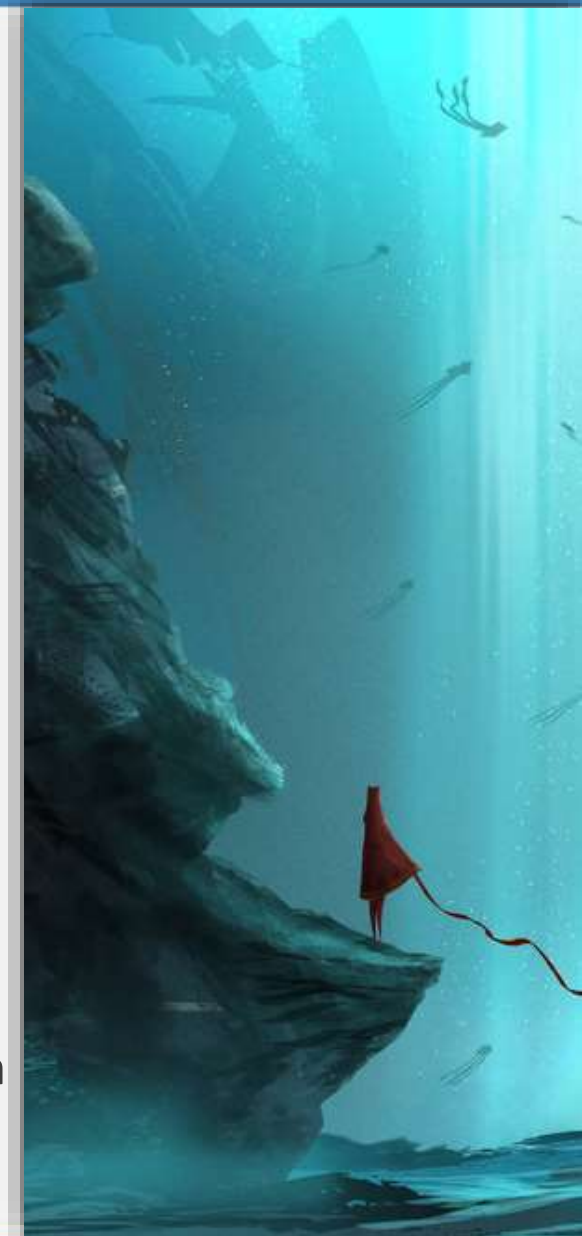
```
Inside function: sub_0005258 (0x0005258)  
struct:USART1 address: 0x00052ba1 (writing to register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (writing to register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (writing to register)  
struct:USART1 address: 0x00052ba1 (reading from register)  
struct:USART1 address: 0x00052ba1 (reading from register)
```





5. LOS TUNELES — DEBUGGING ARM

- Usualmente el análisis estático de las funciones no es suficiente
 - O es demasiado costoso
- El problema del debugging
 - Múltiples procesadores
 - Conexiones difíciles
 - Condiciones de carrera
 - ...
- Trucos para debugger firmware en tu PC
 - mmap
 - source code copy
 - Baremetal emulation





5. LOS TUNELES — DEBUGGING ARM

- Restricciones
 - A priori no es posible hacer entrada / salida
 - Ni interactuar con los periféricos
 - No es posible interactuar con zonas de memoria que puede que no se hayan inicializado
 - Mejor para estudiar funciones aisladas





5. LOS TUNELES — DEBUGGING ARM

- Offline debugging - mmap

1. Abrimos el fichero
2. Mapeamos con mmap a la función que nos interese
3. Creamos un puntero a función a la dirección que queremos probar
4. Llamamos a la función con argumentos similares a los que vemos en el código

```
11 int main(void) {
12
13     size_t length=65536;
14     printf("opening file\n");
15     int fd = open("/root/Desktop/STM32f072.bin",0);
16
17     void *firmware=mmap(
18         (void*) 0x08000000, length,
19         PROT_EXEC|PROT_READ|PROT_WRITE,
20         MAP_PRIVATE,          // flags
21         fd,                    // file
22         0                      // offset
23     );
24
25     // Definimos un puntero a función
26     _DWORD* (*functionPtr) (_DWORD*, _BYTE*, unsigned int);
27     functionPtr = 0x08005180; // Apuntamos a nuestra función
28
29     int* address = (int *)0x8008a00; // Nota: address debe existir en
30                                     // la memoria del proceso o SegFault
31
32     printf("Memory address is: 0x%x\n", address);
33     printf("Content of that address is: 0x%x\n", *address);
34
35     _BYTE *arg1 = malloc(32 * sizeof(_BYTE));
36     memset(arg1, 'A', 32);
37     (*functionPtr +1)((_DWORD*) arg1, (_BYTE *) 0x08008a00, 0x18);
38
39     for(int i=0;i<32;i++) printf("%02x ", arg1[i]);
40     printf("\n");
41
42     return 0;
43 }
```





5. LOS TUNELES — DEBUGGING ARM

- Offline debugging – source code copy
 1. Decompilamos la función que queremos estudiar con Hex Rays
 2. Copiamos la función a nuestro archive C
 3. Llamamos a la función con los argumentos apropiados

```
15  DWORD *sub_8005180(_DWORD *result, _BYTE *a2, unsigned int a3)
16  {
17      int v3; // r3@3
18      char v4; // cf@7
19      int result_;
20      if ( !(((int)result | (int)a2) << 30) )
21      {
22          while ( a3 >= 4 )
23          {
24              v3 = *a2;
25              a2 += 4;
26              a3 -= 4;
27              *result = v3;
28              ++result;
29          }
30      }
31      while(a3>=1)
32      {
33          *result = *a2;
34          result = (result + 1);
35          ++a2;
36          a3--;
37      }
38
39      return result;
40  }
41
42  int main(void) {
43      _BYTE *arg1 = malloc(32 * sizeof(_BYTE));
44      memset(arg1, 'A', 32);
45      int returned = sub_8005180(arg1, 0x08008a00, 0x18);
46      for(int i=0;i<32;i++) printf("%02x ", arg1[i]);
47      printf("\n");
48      return 0;
49  }
```





5. LOS TUNELES — DEBUGGING ARM

- **Offline debugging – compilar**
 - Compilar con:
 - `arm-linux-gnueabi-gcc -static sccopy-trick.c -o sccopy-trick -g`
 - Ejecutar con:
 - `qemu-arm sccopy-trick`
 - Depurar con:
 - `qemu-arm -singlestep -g 1234 mmap-trick`
 - `gdb-multiarch`
 - `(gdb) target remote localhost:1234`





5. LOS TUNELES — DEBUGGING ARM

- **Offline debugging – baremetal emulation**
 - GNU MCU Eclipse es un proyecto de código abierto que incluye una familia de plug-ins y herramientas Eclipse para el desarrollo de ARM y RISC-V embebidos multiplataforma, basados en la toolchain GNU. Este proyecto está alojado en GitHub.
 - <https://gnu-mcu-eclipse.github.io/>
 - GNU MCU Eclipse QEMU plugin
 - Puede ser usado de manera independiente
 - Soporta bastantes dispositivos (incluyendo STM32)
 - ... y placas de desarrollo





5. LOS TUNELES — DEBUGGING ARM

- Offline debugging – baremetal emulation

- Ejecutar qemu

- `./qemu-system-gnueabi -mcpu STM32F103RB -nographic --image /root/Desktop/STM32F103.bin -verbose -serial pty -serial pty -serial pty -serial pty -S -s`

- Ejecutar gdb

- `gdb-multiarch`
 - `(gdb) target remote localhost:1234`

- Monitorizar puertos serie

- `tail -f /dev/pts/? | hexdump -C`

```
GNU gdb (Debian 7.12-6+b1) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0800020c in ?? ()
(gdb)
```

Reset ISR





6. EL TEMPLO — FUZZ TESTING

- Hasta ahora tenemos
 - Acceso a SWD / JTAG
 - Conocimiento de nuestro objetivo a través del reversing
 - Estático (IDA Pro)
 - Dinámico offline (debugging)
 - Dinámico (bus eavesdropping)
 - ¡Es hora de encontrar bugs! ¡Fuzzing!





6. EL TEMPLO — FUZZ TESTING

- Debemos de ser capaces de, de manera automática:
 - Generar datos de prueba
 - Transmitirlos a nuestro objetivo
 - Monitorizar y registrar el comportamiento



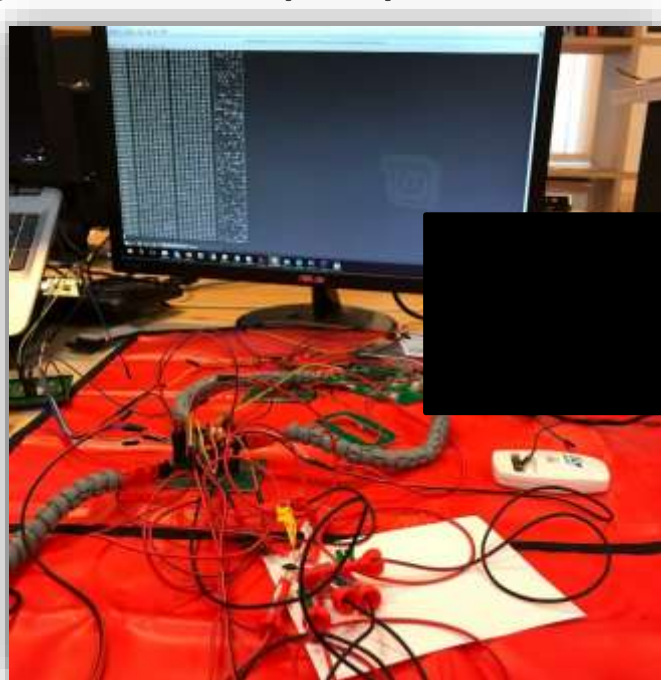
- Depende de la interfaz. En nuestro caso (entre otras) interfaces SAM





6. EL TEMPLO — FUZZ TESTING

- Conectamos a nuestro micro con las interfaces que hemos externalizado usando un programador propio de la marca (recordad que JTAG no es del todo estándar).





6. EL TEMPLO — FUZZ TESTING

```
webdev@webdev-virtual-machine ~/Descargas/openocd-0.10.0 $ sudo openocd -c "telnet_port 4444" -f tcl/interface/stlink-v2.cfg -f tcl/target/stm32f1x_stlink.cfg
Open On-Chip Debugger 0.9.0 (2015-09-02-10:42)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
WARNING: target/stm32f1x_stlink.cfg is deprecated, please switch to target/stm32f1x.cfg
Info : auto-selecting first available session transport 'hla_swd'. To override use 'transport select <transport>'.
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 1000 kHz
adapter nrst_delay: 100
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v29 API v2 SWIM v7 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.809987
Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

```
webdev@webdev-virtual-machine ~/Descargas/openocd-0.10.0 $ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
```

```
> halt
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x08005cfc msp: 0x20004730
```

targets	TargetName	Type	Endian	TapName	State
0*	stm32f1x.cpu	hla_target	little	stm32f1x.cpu	halted

- Conectamos adecuadamente
- Arrancamos OpenOCD
- Ejecutar gdb
 - gdb-multiarch
 - (gdb) target remote localhost:3333
- ¡Lanzamos fuzz-testing y monitorizamos!





7. LA MONTAÑA – EXPLOTACION

- Aplica todo lo que ya conocéis
 - Format string bugs
 - Stack overflow
 - ...
- No hay protecciones del S.O.
 - No ASLR
- ¡Con una interesante diferencia!

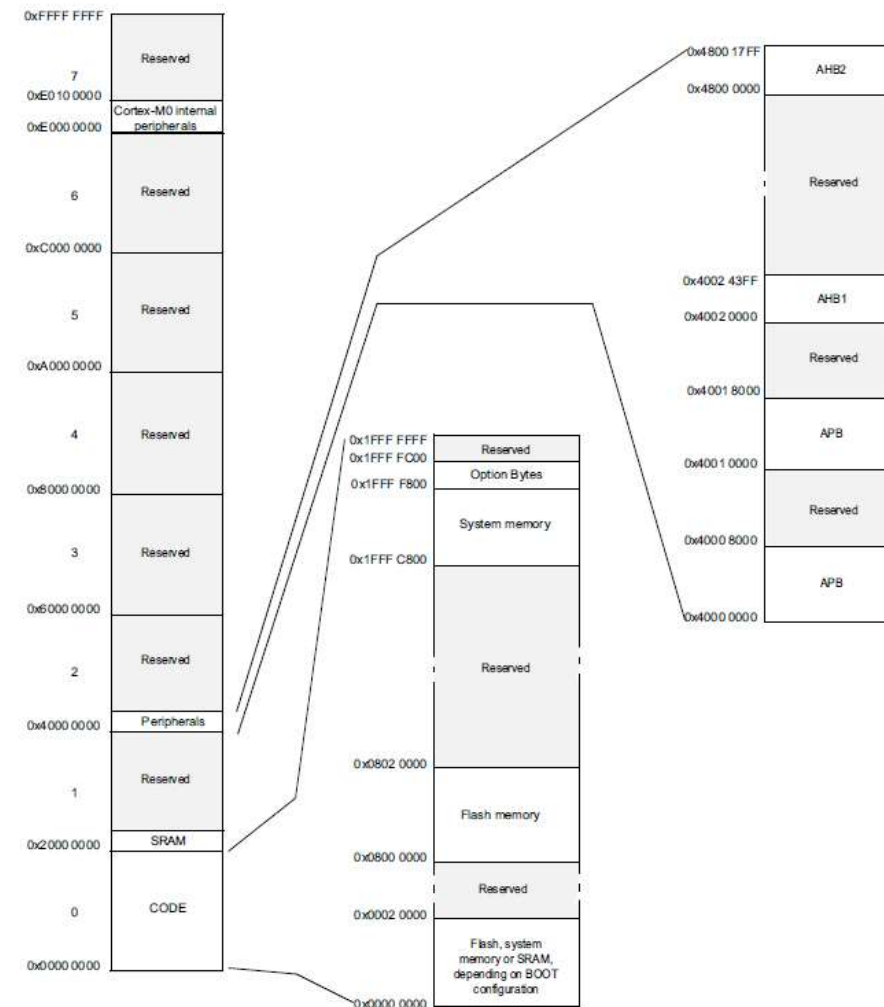




7. LA MONTAÑA – EXPLOTACION

- ¿Qué ocurre al leer la dirección 0x00000000?
 - Windows → Crash (Segmentation Fault)
 - Linux → Crash (Segmentation Fault)
 - Baremetal → Depende
 - Usualmente no hay MMU
 - ¡En 0x00000000 hay una copia del firmware!

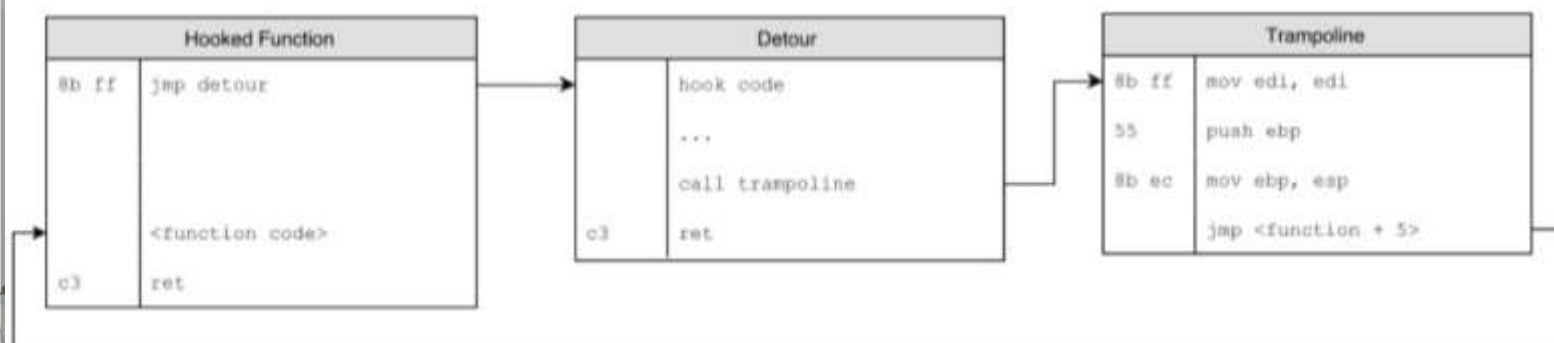
Figure 10. STM32F072xB memory map





8. LA CUMBRE — MANTENIENDO EL ACCESO

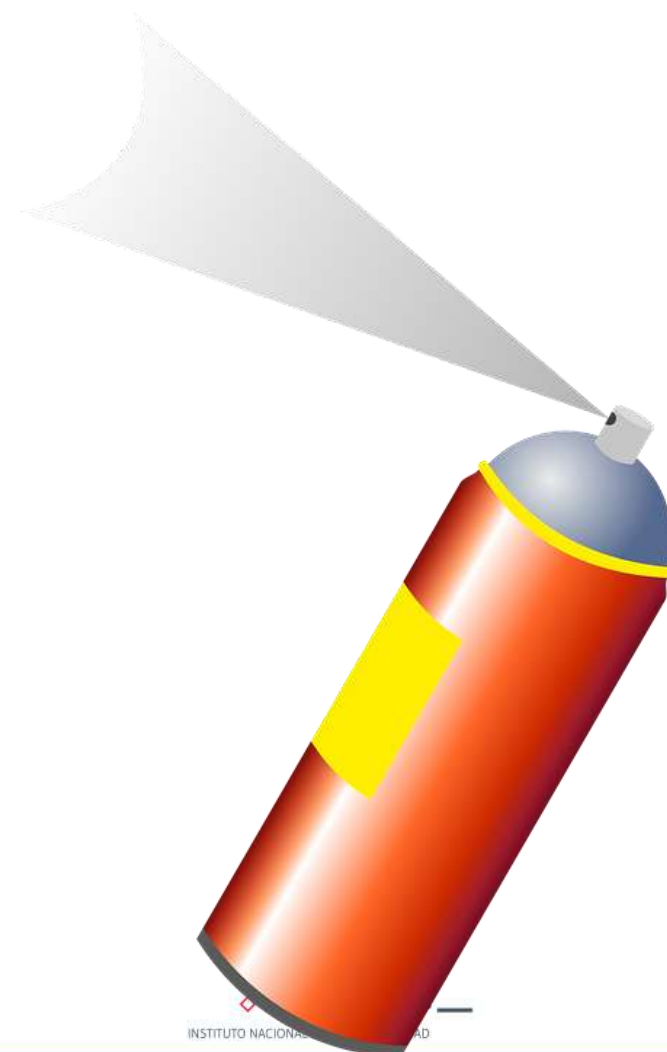
- Trojanizar un firmware baremetal es relativamente sencillo!
- **Flash:**
 - Concatenamos nuestro código al código original
 - Modificamos el código original para que salte al nuestro





B. LA CUMBRE — MANTENIENDO EL ACCESO

- Necesitaremos espacio para nuestras variables
 - **SRAM:**
 - Pintamos la SRAM con un valor conocido antes de que se ejecute el código.
 - Dejamos que el dispositivo arranque e interactuamos con él.
 - Volcamos de nuevo la SRAM para encontrar regiones no utilizadas.
 - Cuando sepa qué regiones de memoria están disponibles, puede decirle a su compilador que use estas regiones de memoria.





CONCLUSIONES

- Lee las especificaciones!
- Asume que el firmware es público
- Protege los buses!
- Cifra! Siempre!
- ¿Physical access == Game Over?



mmisheearthblogs.tumblr.com
mmishee.deviantart.com





jtsec: Beyond IT Security

c/ Abeto s/n Edificio CEG Oficina 2B

CP 18230 Granada – Atarfe – Spain

hola@jtsec.es

@jtsecES

www.jtsec.es

¡GRACIAS POR AGUANTAR!

