UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Lecture 5 - Exploiting Low Level Communications

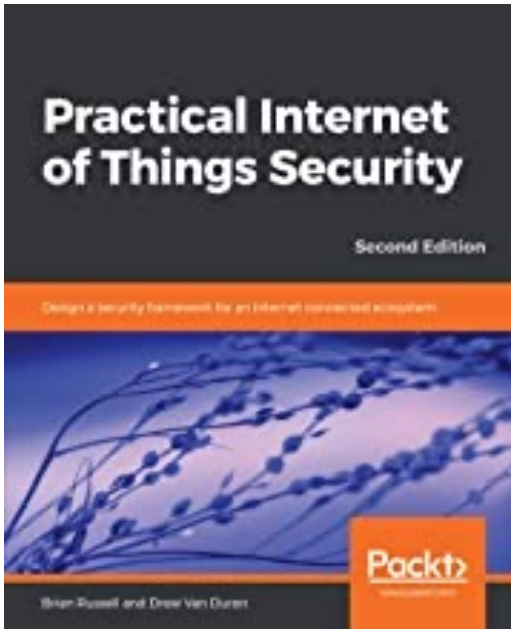
Prof. Esposito Christian

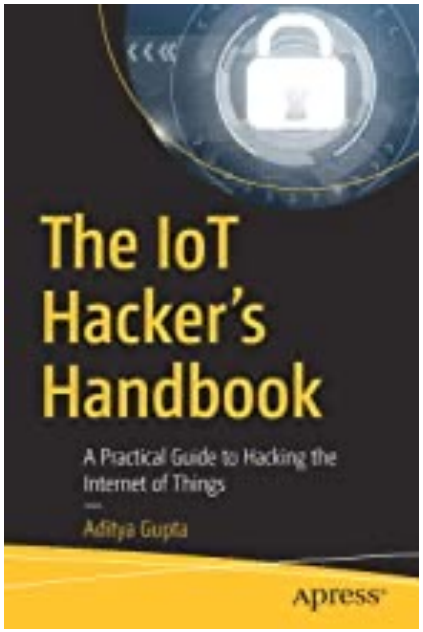
::.. Summary

- Tampering IoT Nodes – Part 1
  - Node Capture
  - Low-level Communication Means
  - Exploitation of the Low-level Communication Protocols
  - JTAG and its Exploitation
  - Protecting Low-Level Communications

::.. References

- Brian Russell, Drew Van Duren, “Practical Internet of Things Security: Design a security framework for an Internet connected ecosystem”, Packt Publishing; 2nd edition - November 2018;
- Aditya Gupta, “The IoT Hacker's Handbook: A Practical Guide to Hacking the Internet of Things”, Apress - April 2019.





::.. Key Lectures

- Becher, Alexander, Zinaida Benenson, and Maximillian Dornseif. "Tampering with motes: Real-world physical attacks on wireless sensor networks." International Conference on Security in Pervasive Computing, 2006.
- Yang, Kaiyuan, David Blaauw, and Dennis Sylvester. "Hardware designs for security in ultra-low-power IoT systems: an overview and survey." IEEE Micro 37.6 (2017): 72-89.
- Parameswaran, Sri, and Tilman Wolf. "Embedded systems security—an overview." Design Automation for Embedded Systems 12.3 (2008): 173-183.

Tampering IoT Devices

::.. Node Capture

One of the possible attacks to the devices composing the perception layer of an IoT is the **node capture**. It describe the case of an adversary gaining full control of IoT devices through direct physical access.

> This is fundamentally different than gaining control remotely trough a software bug. As all the devices typically run the same software, if a software bug is found and exploited, the adversary is able to control all the devices running the same bugged software.

A node capture due to tampering throughout a physical access can be conducted against a small portion of the sensor network.

The impact of this attack can be marginal if negligible information is stolen, or greater if routing, data provisioning and/or other critical capabilities are affected.

::.. Info Acquisition (1/4)

Running a node capture attack requires a precise knowledge of the victim device, so as to find the available physical interfaces to access the hardware.

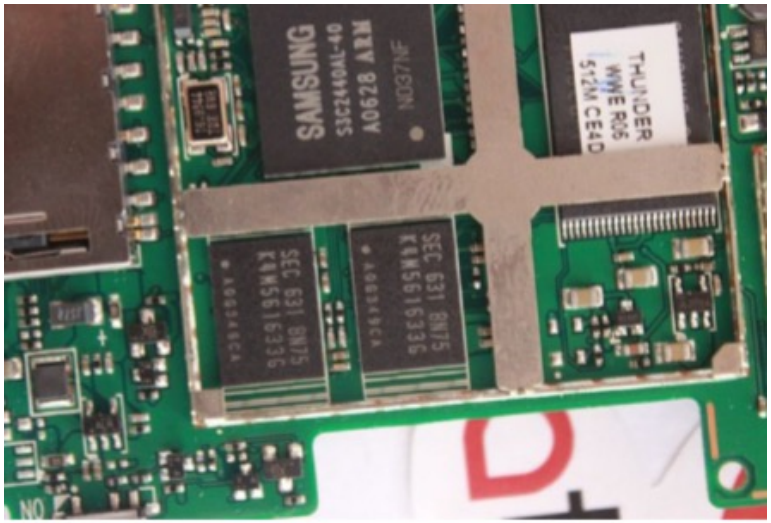
Unless the wanted information may be accessible through the Internet, or obtainable hardware specification, an external visual inspection is performed, by looking at how the I/O of the device works, which interfaces are exposed to the outside.

In addition, the device may be used in order to acquire its functional description in normal conditions.

Afterwards, an internal inspection is performed by opening it up and looking at its internal components. It is a delicate operation as the device may be damaged at the end, or it may be hard to unscrew the screws.

::.. Info Acquisition (2/4)

These actions allows the adversary to come up with a target approach to tamper the device, by altering its software, hold data and so on. At the end the attack surface of the device by exploiting the vulnerability of deficient physical security.



As an example, by internal inspection we can find the use of a Samsung ARM processor whose identifier is S3C2440AL, and by looking the datasheet on line, it is possible to obtain valuable details. The same is possible with the other components such as SDRAM and ROM.

::.. Info Acquisition (2/4)

These actions allows th t approach to tamper th d data and so on. At the y exploiting the vulnerabil

FEATURES

Architecture

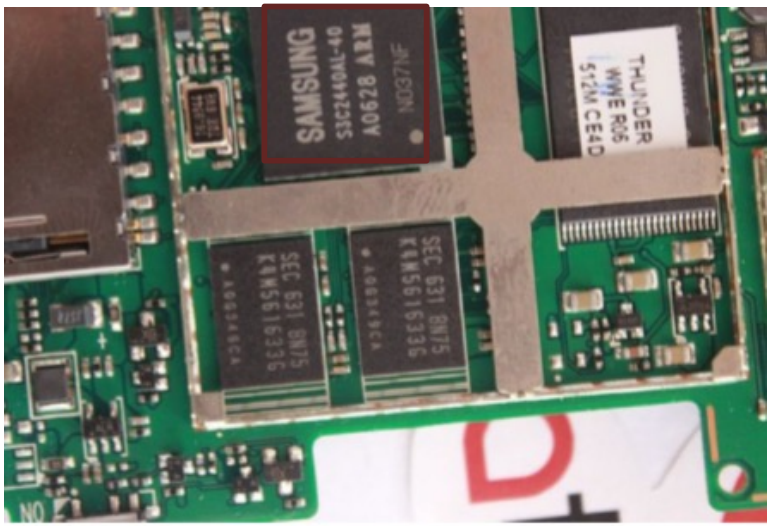
- Integrated system for hand-held devices and general embedded applications.
- 16/32-Bit RISC architecture and powerful instruction set with ARM920T CPU core.
- Enhanced ARM architecture MMU to support WinCE, EPOC 32 and Linux.
- Instruction cache, data cache, write buffer and Physical address TAG RAM to reduce the effect of main memory bandwidth and latency on performance.
- ARM920T CPU core supports the ARM debug architecture.
- Internal Advanced Microcontroller Bus Architecture (AMBA) (AMBA2.0, AHB/APB)

NAND Flash Boot Loader

- Supports booting from NAND flash memory.
- 4KB internal buffer for booting.
- Supports storage memory for NAND flash memory after booting.
- Supports Advanced NAND flash

Cache Memory

- 64-way set-associative cache with I-Cache (16KB) and D-Cache (16KB).
- 8words length per line with one valid bit and two dirty bits per line.
- Pseudo random or round robin replacement algorithm.

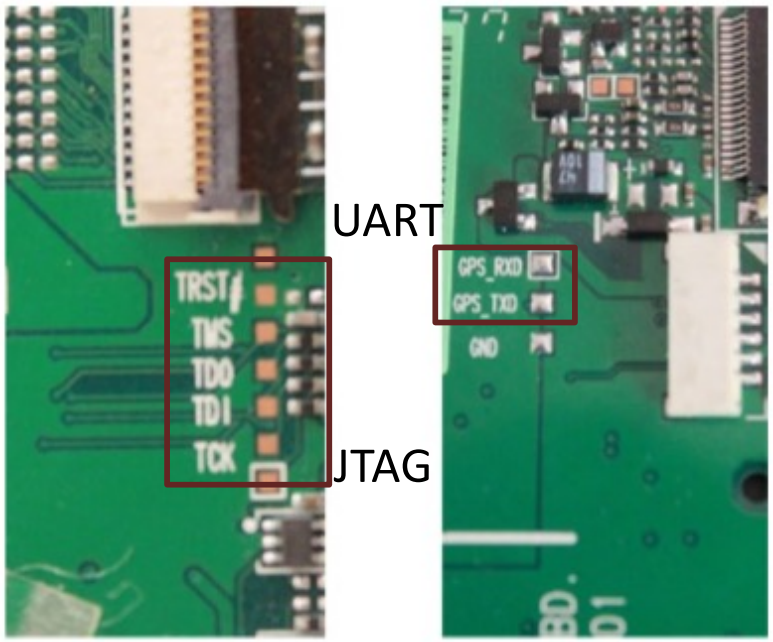


As an example, by internal inspection we can find the use of a Samsung ARM processor whose identifier is S3C2440AL, and by looking the datasheet on line, it is possible to obtain valuable details. The same is possible with the other components such as SDRAM and ROM.

::.. Info Acquisition (3/4)

A particularly important aspect to determine is represented by debug ports and interfaces.

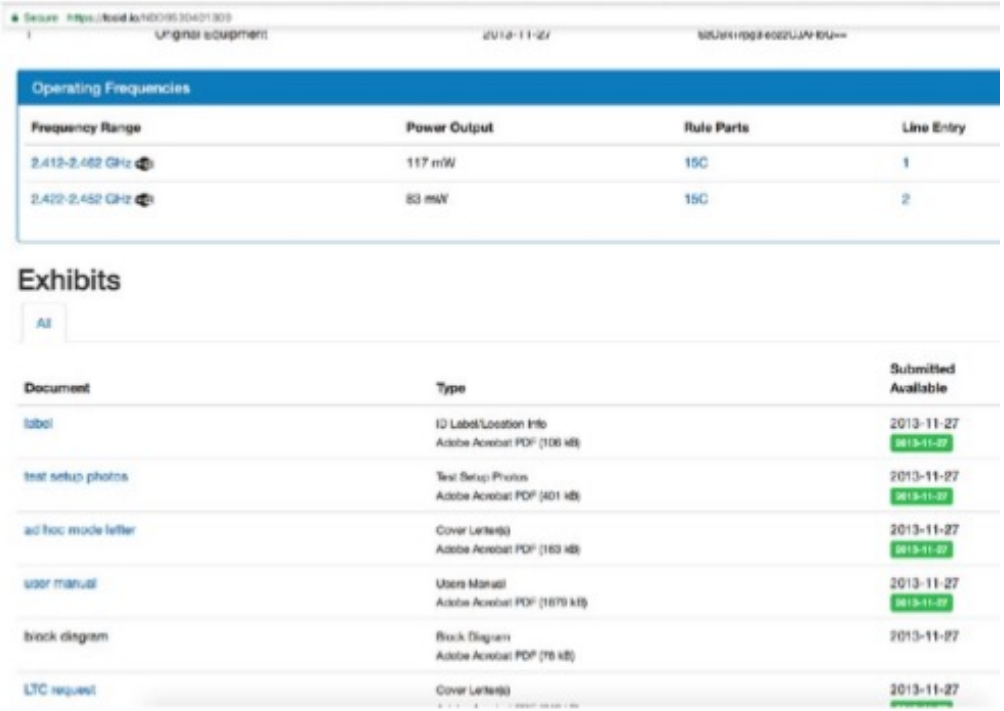
- Devices expose communication interfaces to be exploited to gain access to the device and perform malicious actions, such as gain logs or unauthenticated root shell on the target device.



- Some interfaces are evident externally, but others can only be seen by looking at the PCB and identifying Tx and Rx for UART and TRST, TMS, TDO, TDI and TCK for JTAG.

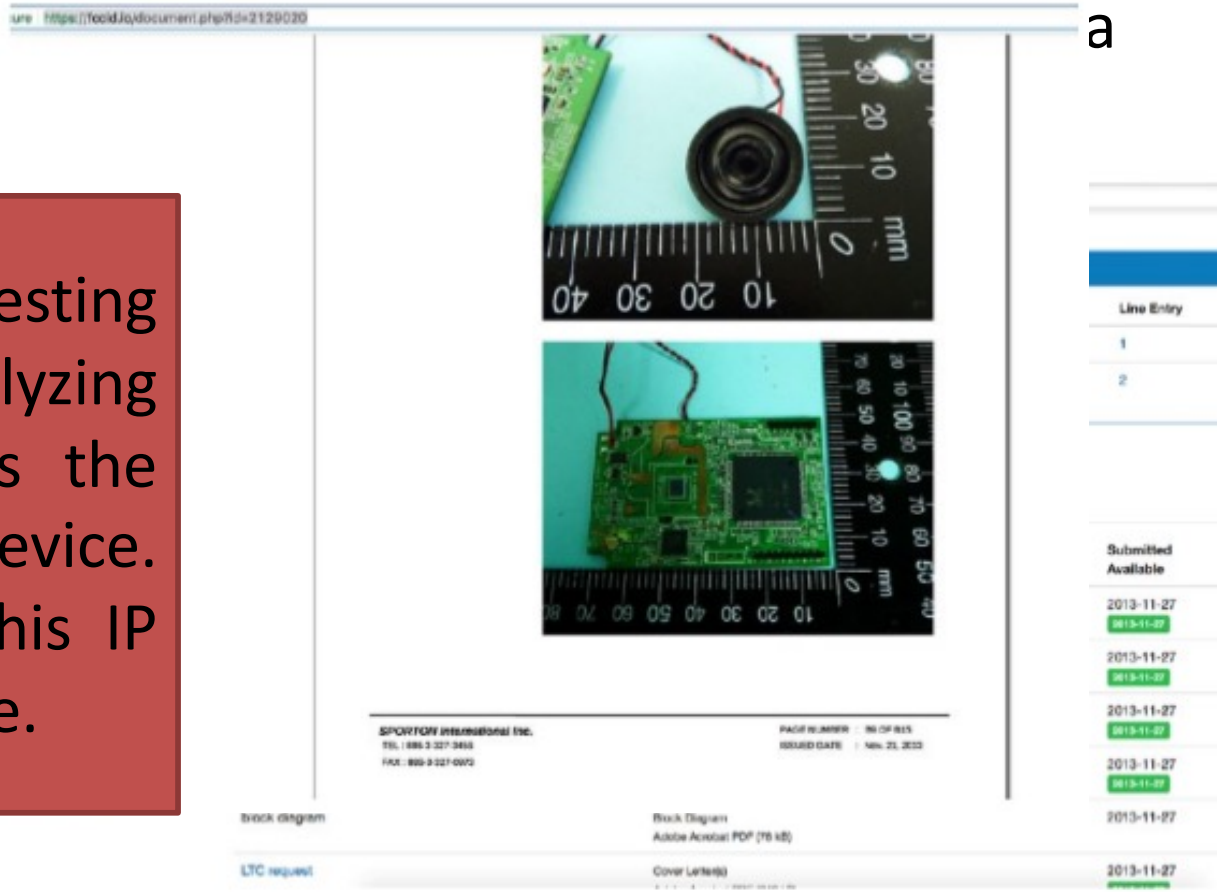
::.. Info Acquisition (4/4)

For certain equipment, on/line is hard to find valuable technical information, but the Federal Communication Commission (FCC) ID may come in help by querying the FCC database located at https://apps.fcc.gov/oetcf/eas/reports/GenericSearch.cfm or via a third-party unofficial web site such as fccid.io or fcc.io.



::.. Info Acquisition (4/4)

For certain equipment, on/line is hard to find valuable technical information, but the Federal Communication Commission (FCC) ID may come in help by querying the FCC database located at https://apps.fcc.gov/oet(a third-party unofficial w



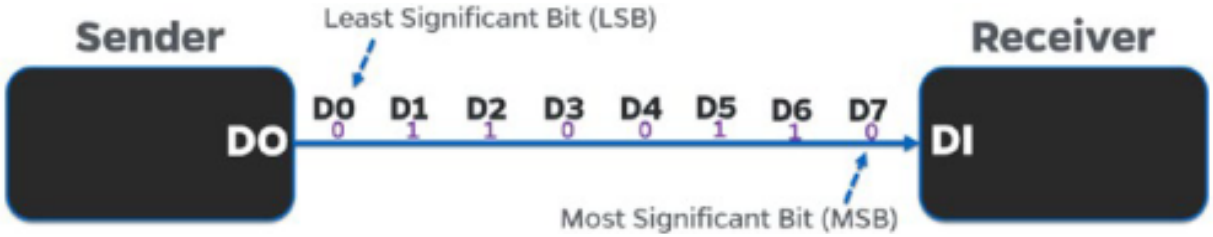
One of the most interesting things to look at while analyzing the FCC ID information is the internal pictures of the device. The pictures reveal that this IP camera has a UART interface.

UART, I^2C, SPI

::.. Low-Level Communication (1/2)

For any embedded device, the different components need to interact with each other and exchange data. Serial communication and parallel communication are the two ways in which components on a device exchange data.

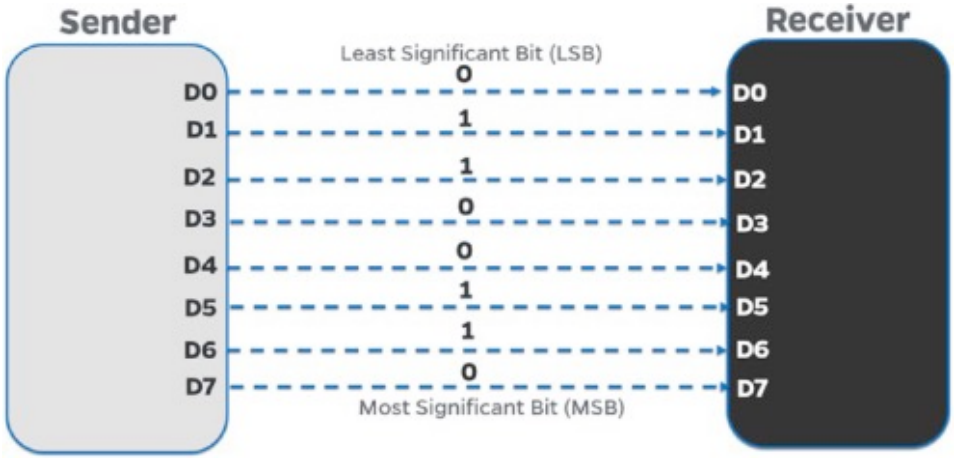
- Serial communication is used to transfer one bit at a time through a given medium;



::.. Low-Level Communication (1/2)

For any embedded device, the different components need to interact with each other and exchange data. Serial communication and parallel communication are the two ways in which components on a device exchange data.

- Serial communication is used to transfer one bit at a time through a given medium;
- Parallel communication consists in transferring a block of data at the same time with each bit requiring a separate channel.



::.. Low-Level Communication (1/2)

For any embedded device, the different components need to interact with each other and exchange data. Serial communication and parallel communication are the two ways in which components on a device exchange data.

- Serial communication is used to transfer one bit at a time through a given medium;
- Parallel communication consists in transferring a block of data at the same time with each bit requiring a separate channel.

Parallel communication is faster, but also require more space occupation on the board. That is the reason serial communication is a more common method: it requires just a single line to facilitate the data exchange.

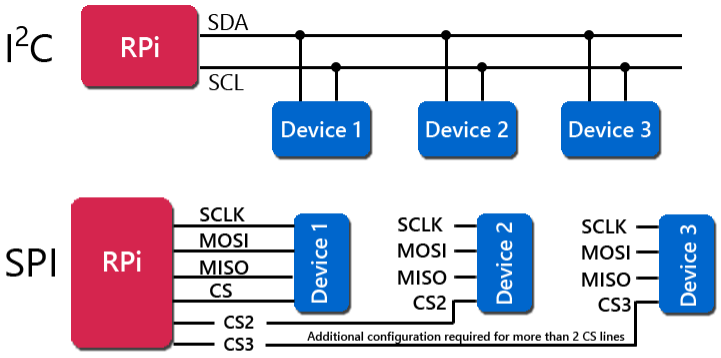
::.. Low-Level Communication (2/2)

Some of the popular serial communication channels you might have heard are Recommended Standard 232 (RS232), Universal Serial Bus (USB), PCI, High-Definition Multimedia Interface (HDMI), Ethernet, Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C), Controller Area Network (CAN), etc.

Due to recent advancements in technology, serial communication is getting cheaper, faster, and more reliable:

- The first serial communication channel used was RS232, which offered a data transmission rate of 20 kbps;
- Then it came the USB 1.0, offering rates of 12 Mbit/s; followed by USB 2.0, with a speed of 480 mbps;
- Finally, the USB 3.0, with a speed of 5 Gbps is currently adopted.

::.. Main Serial Comm. Protocols



I²C, SPI and UART represent standard communications protocols that are available through the GPIO (General Purpose Input/Output) pins of the main boards.

Name	Description	Function
I²C	Inter-Integrated Circuit	Half duplex, serial data transmission used for short-distance between boards, modules and peripherals. Uses 2 pins.
SPI	Serial Peripheral Interface bus	Full-duplex, serial data transmission used for short-distance between devices. Uses 4 pins.
UART	Universal Asynchronous Receiver-Transmitter	Asynchronous, serial data transmission between devices. Uses 2 pins.

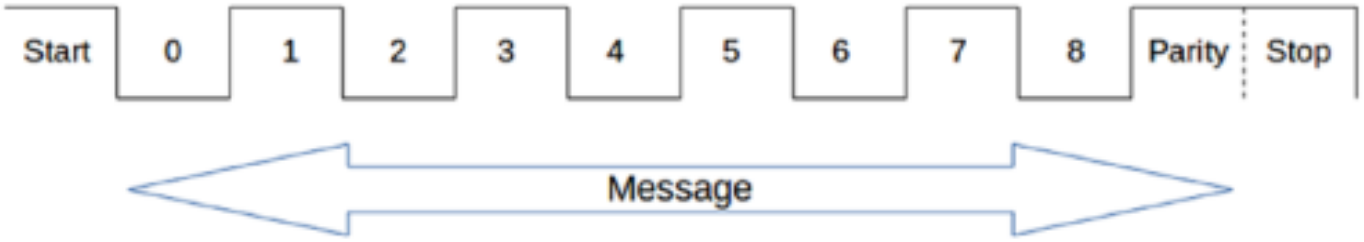
- UART - simple; not high speed; no clock needed; limited to one device connected to the board.
- I²C - faster than UART, but not as fast as SPI; easier to chain many devices; the board drives the clock so no sync issues.
- SPI - fastest of the three; the board drives the clock so no sync issues; practical limit to number of devices on the board.

::.. UART (1/3)

The **Universal Asynchronous Receiver/Transmitter** (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock, i.e., for an additional line of external clock (CLK).

The term asynchronous simply means that unlike a synchronous protocol (e.g., SPI), it does not have a clock that syncs for both the devices between which the communication taking place.

The UART packets structure:



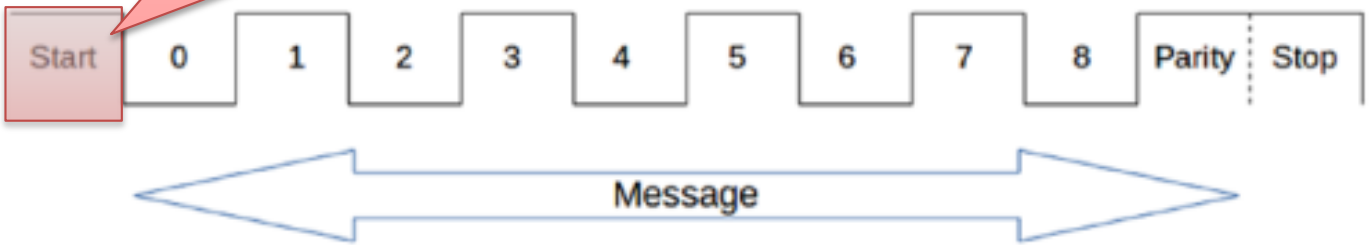
::.. UART (1/3)

The **Universal Asynchronous Receiver/Transmitter** (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock, i.e., for an additional line of external clock (CLK).

The term asynchronous simply means that unlike a synchronous protocol oth the devic

Starting bit: The starting bit symbolizes that the UART data is going to be next. This is usually a low pulse (0) that you can view in the logic analyzer.

The UART packe cture:



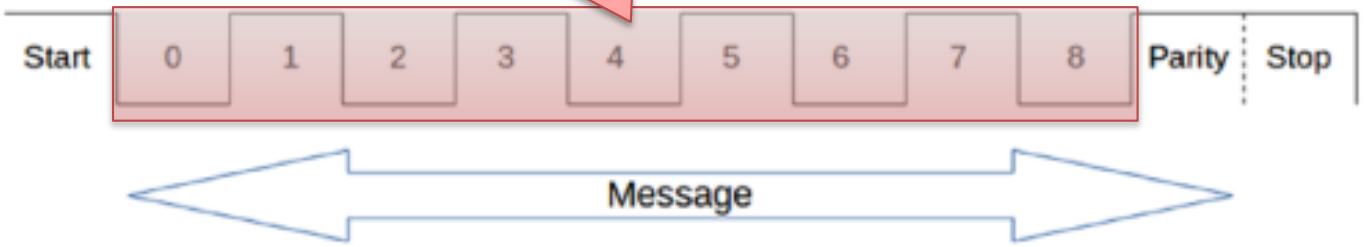
::.. UART (1/3)

The **Universal Asynchronous Receiver/Transmitter** (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock, i.e., for an additional line of external clock (CL

The term ous protocol both the devic

Message: The actual message that is to be transferred as an 8-bit format. For example, if the value A (with the value 0x41 in hex) has to transmit it would be transferred as 0, 1, 0, 0, 0, 0, 0, and 1 in the message.

The UART packets struct



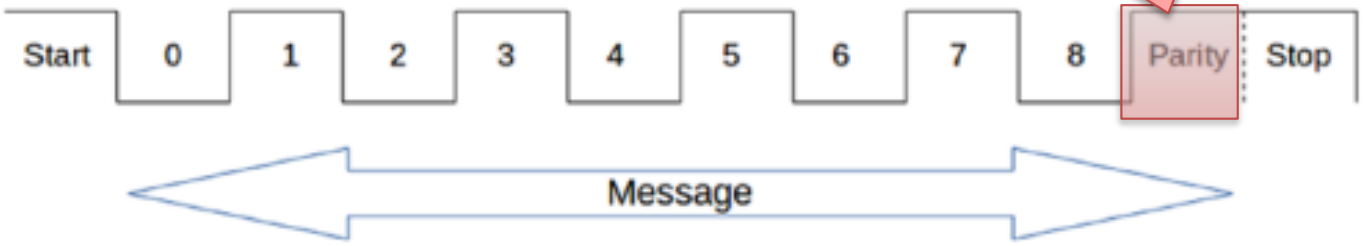
::.. UART (1/3)

The **Universal Asynchronous Receiver/Transmitter** (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock, i.e., for an additional line of external clock (CL

The term ous protocol oth the devic

Parity bit: A bit is used to perform error and data corruption checking by counting the number of high or low values in the message, and based on whether it's an odd parity or an even parity, it would indicate that the data are not correct.

The UART packets structure:



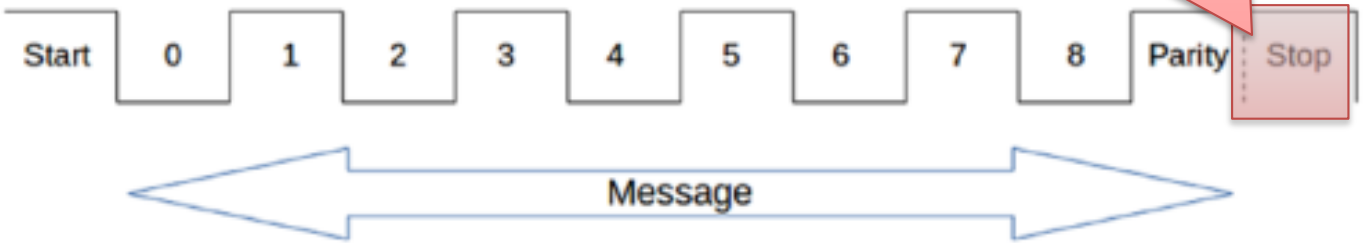
::.. UART (1/3)

The **Universal Asynchronous Receiver/Transmitter** (UART) is a method of serial communication allowing two different components on a device to talk to each other without the requirement of a clock, i.e., for an additional line of external clock (CL

The term ous protocol oth the devic

Stop bit: The final bit that symbolizes that the message has now completed transmission. This is usually done by a high pulse (1), but could also be done by more than one high pulse, depending on the configuration the device developer uses.

The UART packets structure:



::.. UART (2/3)

A UART port could either be hardware based or software based.

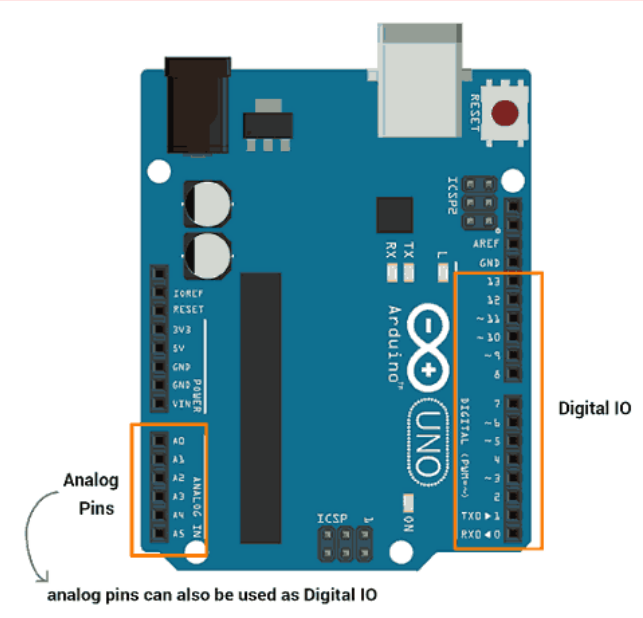
- Atmel's microcontrollers AT89S52 and ATMEGA328 have just one hardware serial port. If it is required, a user is free to emulate more software UART ports on specific general-purpose input/output (GPIOs).
- Microcontrollers like LPC1768 and ATMEGA2560 have multiple hardware UART ports, all of which could be used to perform UART-based analysis and exploitation.

Software-based UARTs are required when there is a need to connect multiple devices via UART to a given device that only has limited sets of hardware UART pins. This also gives the flexibility to the user to use the GPIO pins, as UART when required and use it for another purpose at a later point in time.

::.. UART (2/3)

A UART port could either be hardware based or software based.

- Atmel's microcontrollers AT89S52 and ATMEGA328 have just one hardware serial port. If it is required, a user is free to



A general-purpose input/output (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board whose behavior — including whether it acts as input or output — is controllable by the user at run time.

connect multip a UART to a given device that only has limited sets of hardwar RT pins. This also gives the flexibility to the user to use the GPIO pins, as UART when required and use it for another purpose at a later point in time.

::.. UART (3/3)

The rate at which data are transferred between devices is known as baud rate. This is required to know because there is no clock line in the case of UART communication, so both the communication endpoints need to agree on a single baud rate during the entire UART data exchange process.

During any of the UART exploitation, one of the initial steps will always be to identify the baud rate of the target device. This can be done in several ways, such as looking at the output while interfacing over serial at a given baud rate, and if the data appears to be not readable, moving to the next baud rate.

The common rates are 9600, 38400, 19200, 57600, and 115200 bits per seconds.

::.. UART (3/3)

The rate at which data are transferred between devices is known as baud rate. This is required to know because there is no clock line in the case of UART communication, so both the

To identify the correct baud rate using the approach just mentioned, a script called baudrate.py available at https://github.com/devttys0/baudrate/blob/master/baudrate.py can be used.
This script allows us to change baud rates while maintaining a serial connection, to easily identify what the correct value of the baud rate is by looking at the output and visually inspecting which baud rate gives readable output.

The common rates are 9600, 38400, 19200, 57600, and 115200 bits per seconds.

::.. I^2C & SPI (1/4)

The **Inter-Integrated Circuit** (I^2C) is a multi-master protocol with only two wires required to enable data exchange—serial data (SDA) and serial clock (SCL). However, I^2C is only half-duplex, as it can only send or receive data at a given point of time.

- The challenge with UART is the limitation of facilitating communication between only two devices at a given time.
- A UART packet structure includes a start and stop bit, which adds to the overall size of the data packet that is transferred, also affecting the speed of the entire process.
- UART was originally intended to provide communication over large distances, interacting with external devices via cables. In contrast, I^2C is meant for communicating with other peripherals located on the same circuit board.

::.. I^2C & SPI (2/4)

The **Serial Peripheral Interface** (SPI) is another synchronous bus protocol for communications between different components in an embedded device circuit. It is full-duplex, and consists of three wires — SCK, MOSI, and MISO — and an additional chip select/slave select. Only one single master is controlling all the slaves and the master controls the clock for all the slaves.

SPI has faster data transmission rates compared to I^2C, but the only major downside that SPI has is the requirement of more pins, which increases the overall requirement of space.

Both SPI and I^2C are common protocols when it comes to talking about data storage via Electrically Erasable Programmable Read Only Memory (EEPROM).

::.. I^2C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	Ground
VCC	Power supply
#WP	Write protect
#HOLD	Suspends serial input

Serial EEPROMs typically have eight pins.

::.. I²C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	erial data input
MOSI	a output
GND	
VCC	
#W	
#H	

Serial EEPROMs typically have eight pins.

Because both SPI and I²C (and other protocols) usually have multiple slaves, it is required to be able to select one slave among others for any given action.
When a device is not selected, there will be no communication happening between the master and the slave, and the serial data output pin remains in a high impedance state.

::.. I²C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	ial data output
GND	
VCC	
#W	
#H	

Serial EEPROMs typically have eight pins.

The clock or the SCK (or CLK) pin determines the speed for the data exchange. The master is the one that determines the clock speed that the slaves must adhere to. However, in the case of I²C, the slaves can modify and slow down the clock if the clock speed selected by the master is too fast for the slaves. This process is also known as clock stretching.

::.. I^2C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	
VCC	
#W	
#H	

Serial EEPROMs typically have eight pins.

MISO and MOSI stand for master-in-slave-out and master-out-slave-in, respectively. Depending on who is sending data and who is receiving, the pins are used. In case of I^2C, because it's half-duplex, it can only either read or write data at a given point in time. However, in the case of SPI, both read and write data happens at the same time. If there is no data to be sent (in read or write), dummy data is sent.

::.. I^2C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	Ground
VCC	Power supply
#WP	Write protect
#HOLD	ds serial input

Serial EEPROMs typically have eight pins.

This pin allows normal read/write operations when it is high. When the #WP pin is active low, all write operations are inhibited.

::.. I²C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	Ground
VCC	Power supply
#WP	Write protect
#HOLD	Suspends serial input

Serial EEPROMs typically have eight pins.

When a device is selected and a serial sequence is underway, #HOLD can be used to pause the serial communication with the master device without resetting the serial sequence. To resume the serial sequence, the #HOLD pin should be made high while the SCK pin is low.

::.. I^2C & SPI (3/4)

Pin name	Function
#CS	Chip select
SCK	Serial data clock
MISO	Serial data input
MOSI	Serial data output
GND	Ground
VCC	Power supply
#WP	Write protect
#HOLD	Suspends serial input

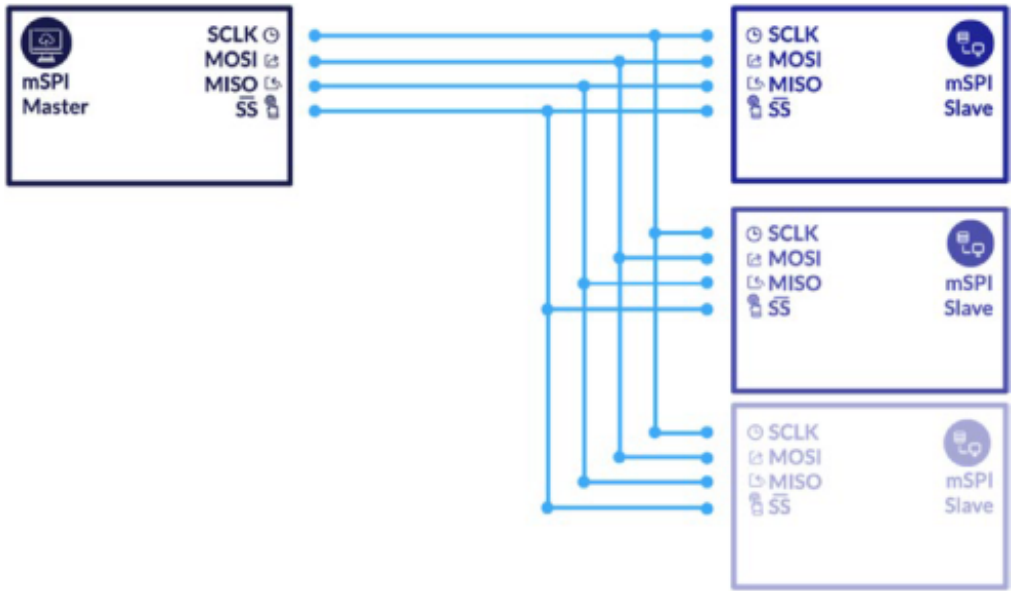
Serial EEPROMs typically have eight pins.

The master also holds the address and memory location of all the various slave devices that are used during any communication.

In I^2C, unlike SPI, there can be multiple masters interacting with various slaves. That configuration is called a multi-master mode.

If two masters wanted to take control over an I^2C bus at the same time, the first one that pulls the SDA (the line for data exchange) to LOW (0) will gain control of the bus.

::.. I^2C & SPI (4/4)



The SCK, MISO, and MOSI pins are shared, whereas each SPI slave will have its own unique SS line (for slave selection).

The master first configures the clock frequency no more than half of the speed of the its clock.

The master starts selecting the slave with a logic level 0 on SS, and initiates by sending a bit on MOSI, being read by the slave, whereas the slave sends a bit on MISO, being read by the master.

The most significant bit (MSB) is shifted out first and a new least significant bit (LSB) is shifted into the same register.

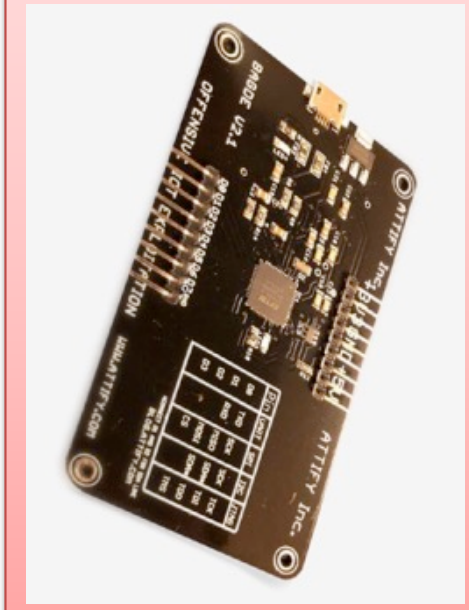
Low Level Communication Exploitation

::.. UART Exploitation (1/7)

To perform a UART-based exploitation, it is needed a device that could emulate a serial connection to access the target device, such as Attify Badge (but a normal USB-TTL or BusPirate can be used as well).

::.. UART Exploitation (1/7)

To perform a UART-based exploitation, it is needed a device that could emulate a serial connection to access the target device, such as Attify Badge (but a normal USB-TTL or BusPirate can be used as well).

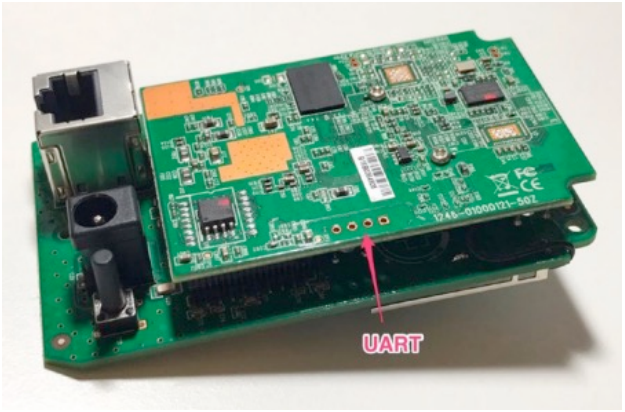


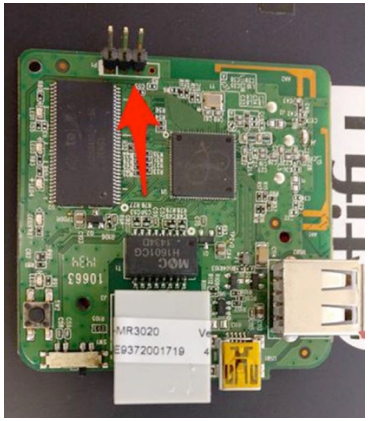
The Attify Badge is a multipurpose tool that helps you communicate to other IoT/embedded devices over various communication interfaces, such as UART, SPI, I^2C, and even standards such as JTAG. It uses an FTDI chip that allows it to convert the hardware communication protocol in a language that is understood by a program running on a traditional computer.

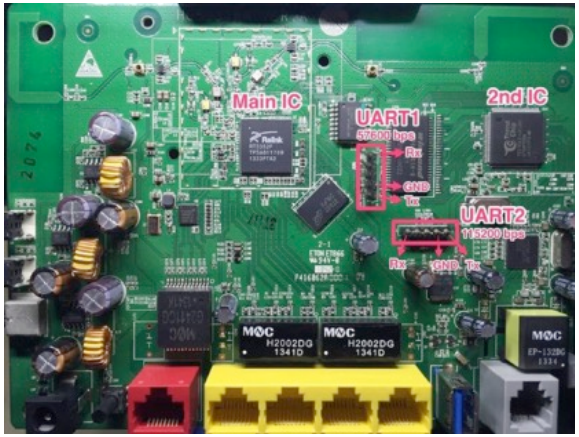
::.. UART Exploitation (1/7)

To perform a UART-based exploitation, it is needed a device that could emulate a serial connection to access the target device, such as Attify Badge (but a normal USB-TTL or BusPirate can be used as well).

To make the connections, it has to be identified where the UART port on the device is, or what the UART pins are. This can be done by a visual inspection of the internal device components and looking for three or four pins or pads close to each other.







::.. UART Exploitation (2/7)

UART consists of four pins:

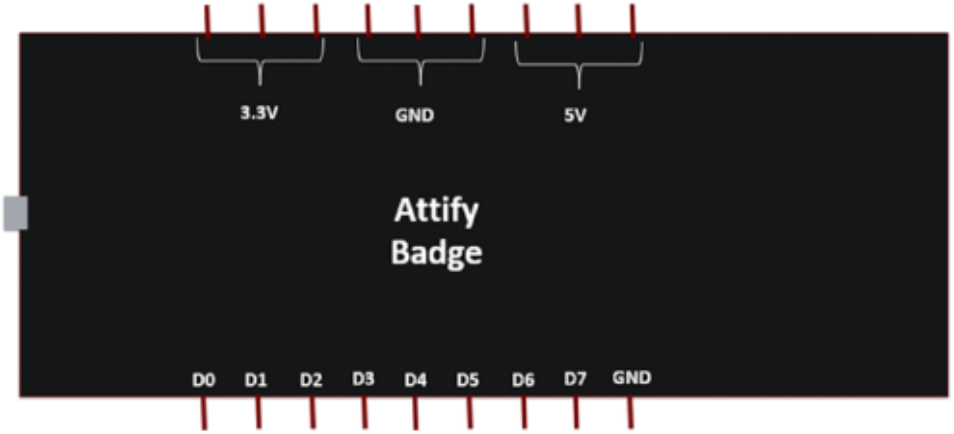
1. Transmit (Tx): Transmits data to the other end;
2. Receive (Rx): Receives data from the other end;
3. Ground (GND): Ground reference pin;
4. Voltage (Vcc): Voltage, usually either 3.3V or 5V.

A multimeter can be used to identify these pins based on either the continuity test (for GND) or by looking at the voltage difference (for the remaining three pins).

A multimeter is a combination of a voltmeter and ammeter, measuring both the value of voltage and the value of current during the analysis.

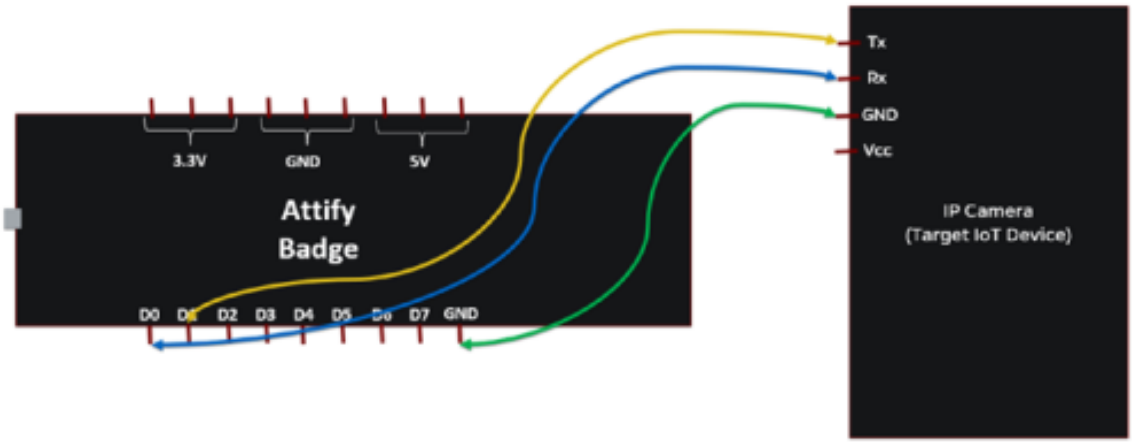
::.. UART Exploitation (3/7)

The Attify Badge contains a total of 18 pins out of which 10 pins are for voltage (3.3V and 5V) and ground, which are the top nine pins and the bottom right pin. The pins D0 through D3 serve special a purpose when it comes to interacting with embedded device hardware according to a given serial communication means.



Pin	UART	SPI	I2C	JTAG
D0	TX	SCK	SCK	TCK
D1	RX	MISO	SDA*	TDI
D2		MOSI	SDA*	TDO
D3		CS		TMS

::.. UART Exploitation (4/7)



The target's transmit (Tx) would go to the Attify Badge's Rx (D1), and vice versa for Rx of the target and Tx(D0) of the Attify Badge.

The GND of the target device would be connected to the Attify Badge's GND. Remember to not connect the Vcc, as doing that would risk permanent damage to the target device.

The Attify Badge can be connected to a given computer by using a micro USB cable. If using AttifyOS or running on Mac OS, no special tools are needed to work with Attify Badge. However, on Windows, it might need to download the FTDI driver available from https://www.ftdichip.com/FTDrivers.htm.

::.. UART Exploitation (5/7)

By connecting the Attify Badge, the new device should have been detected by the OS of the used computer, as an example on Linux/Mac OS this can be found by looking at the entries of /dev/ :

This represents the COM port /dev/ttyUSB0, which is also the default COM port used by baudrate.py.

git clone https://github.com/devttys0/baudrate.git

sudo python baudrate.py

Some gibberish data is shown, various values of the baud rate need to be selected before seeing readable characters, revealing that the correct baud rate of the device has been detected.

::.. UART Exploitation (6/7)

Once the correct baud rate has been identified, the next step is to interact with the device over UART. This could either be done through the baudrate.py script itself, by pressing Ctrl+C once it detects the correct baud rate, or manually launching a utility, such as screen or minicom, with the identified configurations.

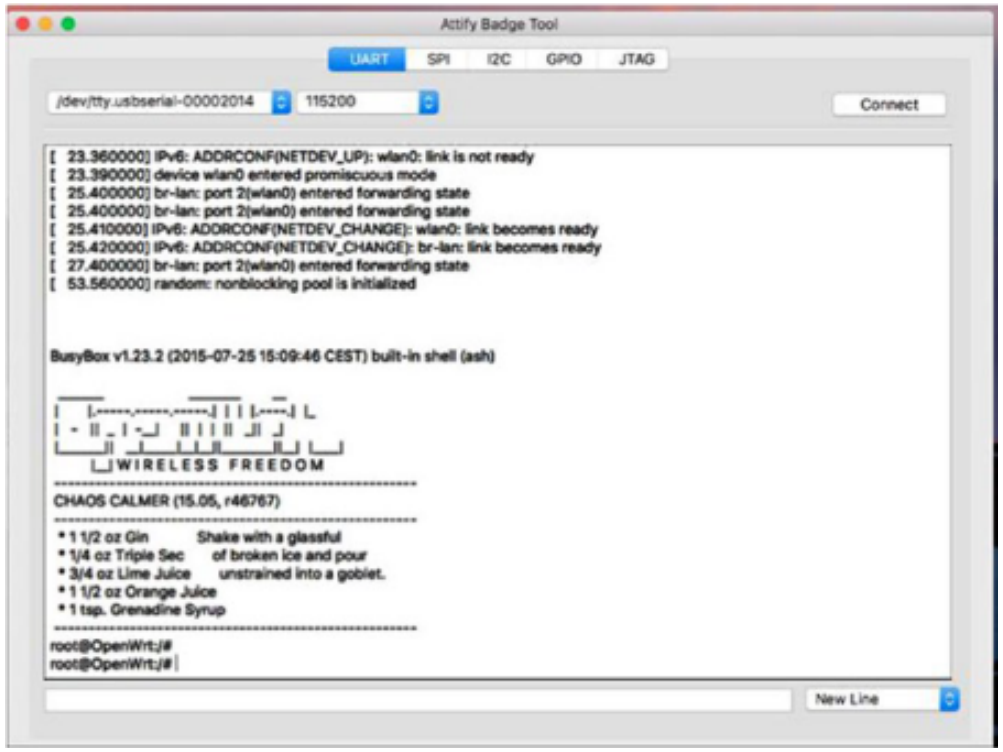
```
Find Port=0 Device:Vender ID=817910ec
vendor_deivce_id=817910ec
=====>>EXIT rtl8192cd_init_one <<=====
=====>>INSIDE rtl8192cd_init_one <<=====
=====>>EXIT rtl8192cd_init_one <<=====


Probing RTL8186 10/100 NIC-kenel stack size order[2]...

Booting...

********************************************************************************
*
* chip_no chip_id mfr_id dev_id cap_id size_sft dev_size chipSize
* 0000000h 0c22017h 00000c2h 0000020h 0000017h 0000000h 0000017h 0800000h
* blk_size blk_cnt sec_size sec_cnt pageSize page_cnt chip_clk chipName
* 0010000h 0000080h 0001000h 0000800h 0000100h 0000010h 000002dh MX25L6405D
*
********************************************************************************
```

By rebooting the target device, the debug logs are shown. After waiting for a couple more seconds, a full unauthenticated root shell on the target device is obtained

```
# ls
bdi             misc            ppp             scsi_host       usb_host
block           mtd             scsi_device     sound           video4linux
firmware        net             scsi_disk       tty
mem             pktcdvd         scsi_generic    usb_endpoint
# Sending discover...
```

::.. UART Exploitation (7/7)

A visual tool is available at https://github.com/attify/attify-badge to perform this process using a graphical user interface (GUI).



::.. I^2C Exploitation (1/4)

Exploiting I^2C means reading or writing data of the device using an I^2C EEPROM in a real-world device. For this, any device that has a flash chip working on the I^2C communication protocol can be used.

MICROCHIP 24AA256/24LC256/24FC256

256K I^2C™ CMOS Serial EEPROM

Device Selection Table

Part Number	Vcc Range	Max. Clock Frequency	Temp. Ranges
24AA256	1.7-5.5V	400 kHz(1)	I
24LC256	2.5-5.5V	400 kHz	I, E
24FC256	1.7-5.5V	1 MHz(2)	I

Note 1: 100 kHz for Vcc < 2.5V.
2: 400 kHz for Vcc < 2.5V.

Features:

- Single Supply with Operation Down to 1.7V for 24AA256 and 24FC256 Devices, 2.5V for 24LC256 Devices
- Low-Power CMOS Technology:
  - Active current 400 uA, typical
  - Standby current 100 nA, typical
- Temperature Ranges:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

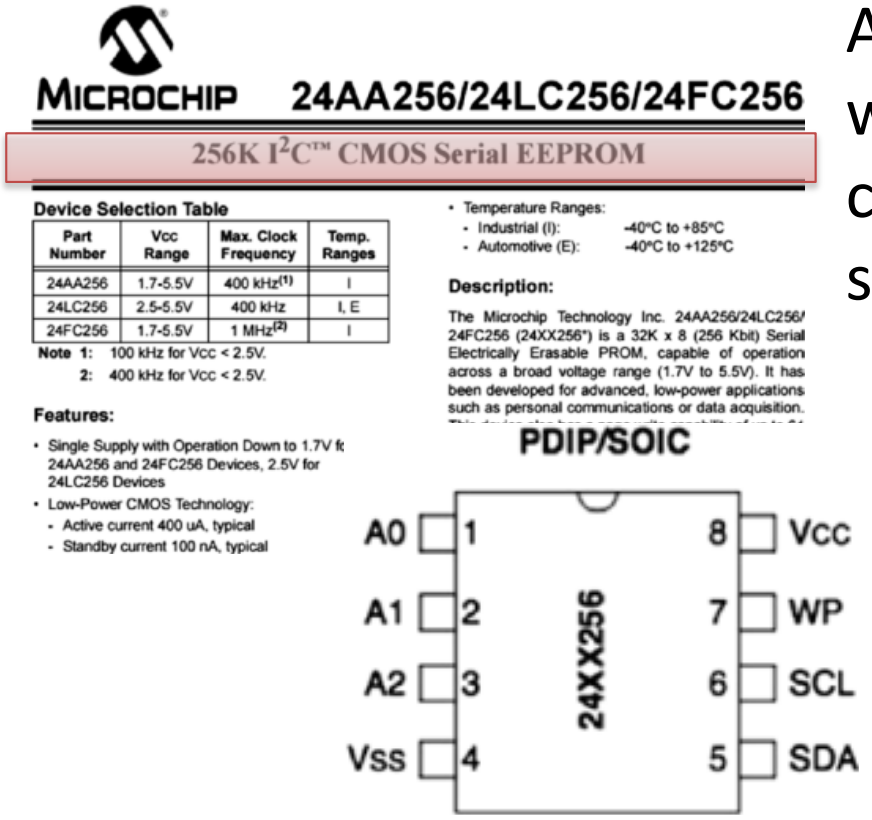
Description:

The Microchip Technology Inc. 24AA256/24LC256/24FC256 (24XX256*) is a 32K x 8 (256 Kbit) Serial Electrically Erasable PROM, capable of operation across a broad voltage range (1.7V to 5.5V). It has been developed for advanced, low-power applications such as personal communications or data acquisition. This device also has a page write capability of up to 64 bytes of data. This device is capable of both random and sequential reads up to the 256K boundary. Functional address lines allow up to eight devices on the same bus, for up to 2 Mbit address space. This device is available in the standard 8-pin plastic DIP, SOIC, TSSOP, MSOP and DFN packages.

A MicroChip 24LC256 EEPROM chip, which works over the I^2C communication protocol, has been selected.

::.. I²C Exploitation (1/4)

Exploiting I²C means reading or writing data of the device using an I²C EEPROM in a real-world device. For this, any device that has a flash chip working on the I²C communication protocol can be used.



MICROCHIP 24AA256/24LC256/24FC256

256K I²C™ CMOS Serial EEPROM

Device Selection Table

Part Number	Vcc Range	Max. Clock Frequency	Temp. Ranges
24AA256	1.7-5.5V	400 kHz(1)	I
24LC256	2.5-5.5V	400 kHz	I, E
24FC256	1.7-5.5V	1 MHz(2)	I

Note 1: 100 kHz for Vcc < 2.5V.
2: 400 kHz for Vcc < 2.5V.

- Temperature Ranges:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

Description:

The Microchip Technology Inc. 24AA256/24LC256/24FC256 (24XX256*) is a 32K x 8 (256 Kbit) Serial Electrically Erasable PROM, capable of operation across a broad voltage range (1.7V to 5.5V). It has been developed for advanced, low-power applications such as personal communications or data acquisition.

Features:

- Single Supply with Operation Down to 1.7V for 24AA256 and 24FC256 Devices, 2.5V for 24LC256 Devices
- Low-Power CMOS Technology:
  - Active current 400 uA, typical
  - Standby current 100 nA, typical

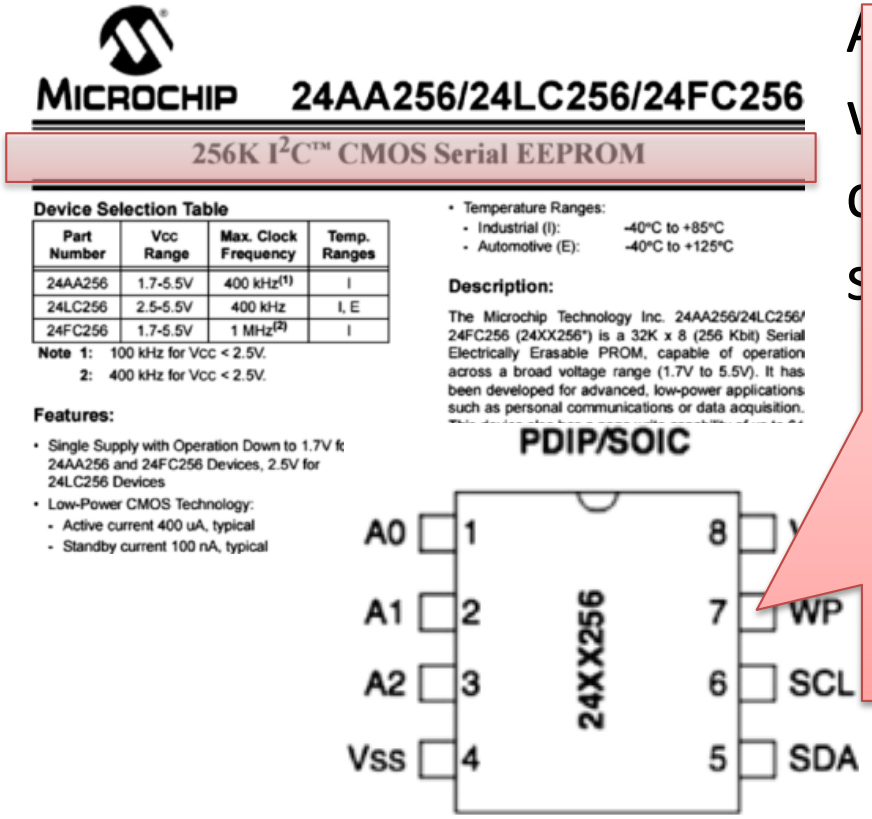
PDIP/SOIC

A MicroChip 24LC256 EEPROM chip, which works over the I²C communication protocol, has been selected.

By finding the component name on the data sheet and looking it up online, a Microchip 256K I²C EEPROM arranged as 32K × 8 serial memory and the pinouts of the EEPROM can be found.

::.. I²C Exploitation (1/4)

Exploiting I²C means reading or writing data of the device using an I²C EEPROM in a real-world device. For this, any device that has a flash chip working on the I²C communication protocol can be used.



MICROCHIP 24AA256/24LC256/24FC256

256K I²C™ CMOS Serial EEPROM

Device Selection Table

Part Number	Vcc Range	Max. Clock Frequency	Temp. Ranges
24AA256	1.7-5.5V	400 kHz(1)	I
24LC256	2.5-5.5V	400 kHz	I, E
24FC256	1.7-5.5V	1 MHz(2)	I

Note 1: 100 kHz for Vcc < 2.5V.
2: 400 kHz for Vcc < 2.5V.

Features:

- Single Supply with Operation Down to 1.7V fo 24AA256 and 24FC256 Devices, 2.5V for 24LC256 Devices
- Low-Power CMOS Technology:
  - Active current 400 uA, typical
  - Standby current 100 nA, typical

- Temperature Ranges:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

Description:

The Microchip Technology Inc. 24AA256/24LC256/ 24FC256 (24XX256*) is a 32K x 8 (256 Kbit) Serial Electrically Erasable PROM, capable of operation across a broad voltage range (1.7V to 5.5V). It has been developed for advanced, low-power applications such as personal communications or data acquisition.

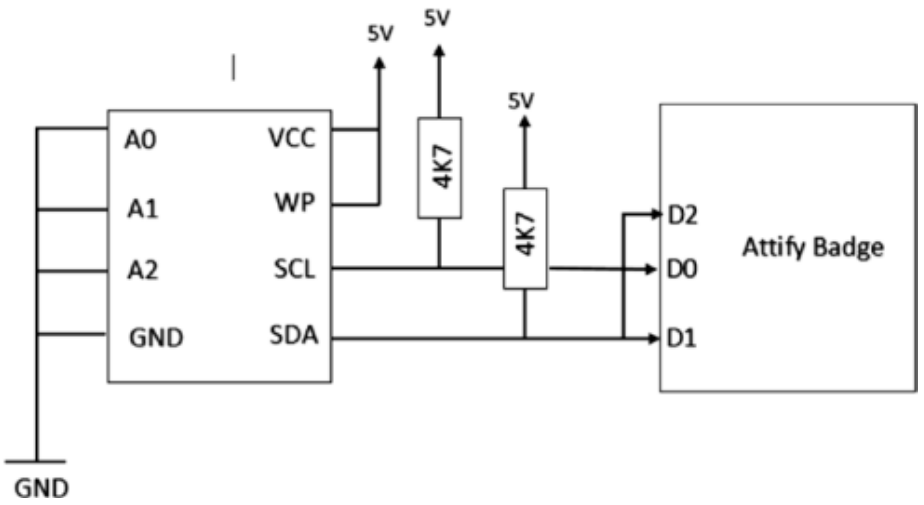
PDIP/SOIC

Pin	Description
A0	User-configurable address bit
A1	User-configurable address bit
A2	User-configurable address bit
VSS	Ground
VCC	1.7V to 5.5V (based on the model)
WP	Write protect (active low)
SCL	Serial clock
SCK	Serial data

…OM chip, …ne I²C …as been …t name on …king it up …256K I²C …× 8 serial memory and the pinouts of the EEPROM can be found.

::.. I²C Exploitation (2/4)

The EEPROM has to be connected to Attify Badge: either by directly connecting it to the Attify Badge by holding the EEPROM using a SOIC clip, or by removing the EEPROM from the device and soldering it on an EEPROM adapter corresponding to the packaging of EEPROM.



- A0, A1, A2, and GND are connected to GND;
- Vcc and WP are connected to 5V, as write protect is active low;
- D1 and D2 of Attify Badge are connected to the SDA line.
- D0 is connected to the I²C SCL (Clock) line.

::.. I^2C Exploitation (3/4)

To work with I^2C, the script i2ceeprom.py located at https://github.com/devttys0/libmpsse/blob/master/src/examples/i2ceeprom.py can be used:

- The script specifies the size of the EEPROM chip, which in this case is 32 KB followed by the Read and Write commands. It also later specifies the speed to be 400 KHz, as shown in the data sheet. Different I^2C EEPROMs might have different speeds and a change of this value is needed.
- The script starts the I^2C clock and sends the Start command to initialize the EEPROM.
- If data have to be read, the script first checks if the EEPROM is available after starting. It then sends the Read command and sets the EEPROM to read mode. Once everything is set, it simply starts reading content from the EEPROM and saving it.

::.. I^2C Exploitation (4/4)

- Once the read operation is complete, the I^2C connection is closed and the saved data is written in a file named EEPROM.bin.

```
root@oit:          /libmpsse/src/examples# python i2ceeprom.py
 FT232H Initialized at 400KHZ
 2493000 bytes dumped to eeprom.bin
```

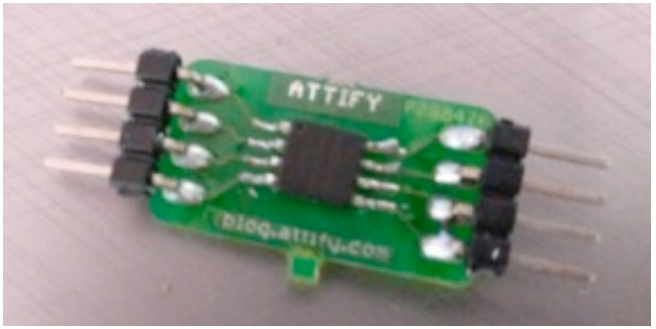
- Similarly, data can be written to the I^2C chip.

::.. SPI Exploitation (1/6)

To read and write data from/to an SPI EEPROM, a utility called spiflash.py can be used and downloaded from https://github.com/devttys0/libmpsse/. In the folder src/examples/ there the script spiflash.py.

- First, it defines several default values, such as the read and write command used by most of the chips using SPI.
- Then, the script define a default speed to interact with the target chip over SPI, by default equal to 15 MHz; however, it can be changed by using the -f parameter.
- The script then also sets the WP and HOLD pins as high.
- The script connects to the target chip over SPI using the mpsse library and performs write, read, and erase operations using the flags provided during runtime, and defined earlier in the code (WCMD, RCMD, WECMD, RECMD).

::.. SPI Exploitation (2/6)



The target flash chip can be removed from the PCB via desoldering. Once it is desoldered, it can then be soldered to an EEPROM adapter (or reader).

It is also possible to directly read it while the chip is on the device by hooking miniprobes to the EEPROM or using a SOIC clip of a real-world IoT device without removing the chip from the device.

The next step would be to make the required connections using the Attify Badge or any supported FTDI-based hardware. To figure out where the numbers start for the pins in the real chip, compared to the one in the data sheet, notice the notch in the top left section of the chip and use it to count the pin numbers.

::.. SPI Exploitation (3/6)

Pin on Attify Badge	Functionality during SPI communication
D0	SCK
D1	MISO
D2	MOSI
D3	CS

Connect CLK to SCK (D0), MOSI/DO to MISO (D1), MISO/DI to MOSI (D2), CS to CS (D3), WP, HOLD, and Vcc to 3.3V, GND to GND.

Once all the connections are in place, the spiflash.py script can be launched to try to read data from the SPI EEPROM.

::.. SPI Exploitation (4/6)

```
root@bit:/home/attify/Downloads/libmpsse/src/examples# python spiflash.py -s 5120000 --read-new.bin
FT232H Future Technology Devices International, Ltd initialized at 15000000 hertz
Reading 5120000 bytes starting at address 0x0...saved to new.bin.
root@bit:/home/attify/Downloads/libmpsse/src/examples# strings new.bin
t11C
*.@"
@"  #
Hu`D
H&"\&2_
@333
*.@"
PPbfu
w rA
K- !A@"
``tg
@300
 $A"a
yq2!
Jh@@tb
``4p6
        a"!
u"      "!
Bq      a
3``tg
 @"
B00t2a
B0t2
@300
error magic!
 @ %x
First boot failed, reboot to try backup bin
Backup boot failed.
2nd Boot version : 1.5
  SPI Speed      :
40MHz
26.7MHz
20MHz
10MHz
  SPI Mode       :
DOUT
DOUT
  SPI Flash Size & Map:
4Mbit(256KB+256KB)
```

Given any device, it will be possible to dump the content that the device has been storing in its EEPROM chip.

::.. SPI Exploitation (4/6)

Given any device, it will be possible to dump the content that the device has been storing in its EEPROM chip.

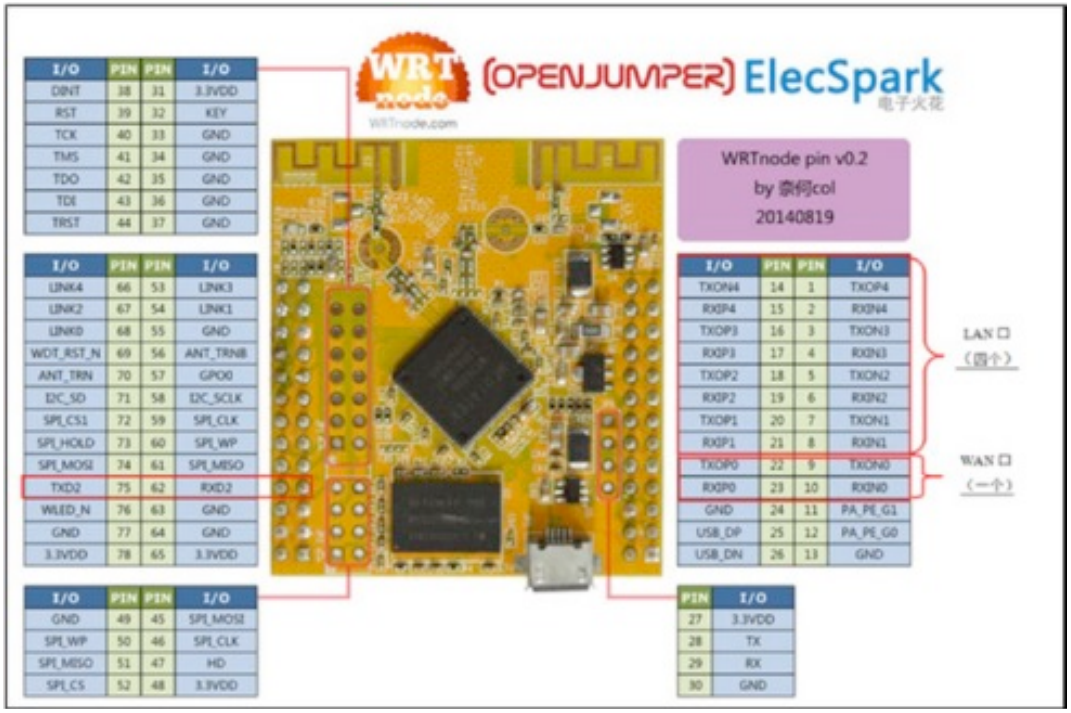
Also data can be written to the chip.

It is possible to write a modified version of a device's firmware if interacting with the EEPROM flash chip over SPI is possible.

::.. SPI Exploitation (5/6)

It’s consider a device named WRTNode having a complete firmware (in this case OpenWRT) and the aim of dumping the firmware using the spiflash.py script and Attify Badge.

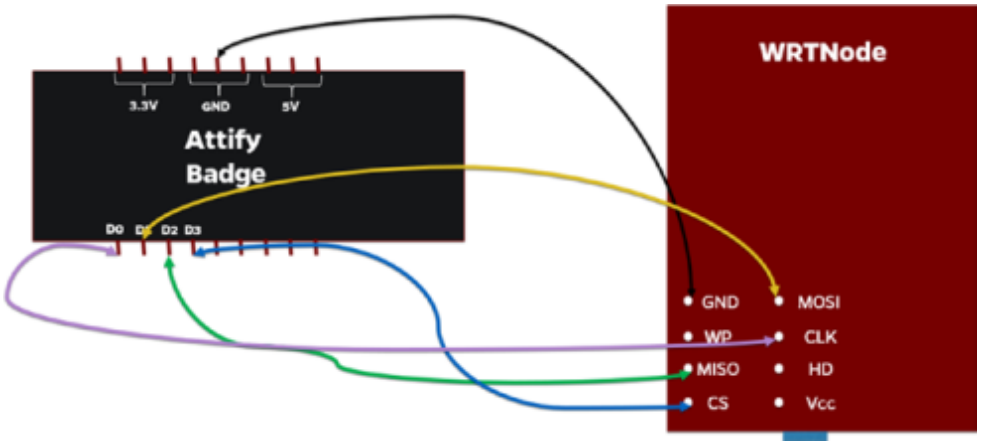
WRTNode has several pins and pads allowing us to connect. The data sheet of WRTNode is available on line because it is a popular development board.



::.. SPI Exploitation (6/6)

WRTNode pin	ATTIFY BADGE
GND	GND
SPI_WP	-
SPI_MISO	D2
SPI_CS	D3 (CS)
SPI_MOSI	D1
SPI_CLK	D0 (CLK)
HD	-
3.3VDD	-

The pins of WRTNode and Attify Badge must be properly connected to support a communication by means of the SPI protocol.

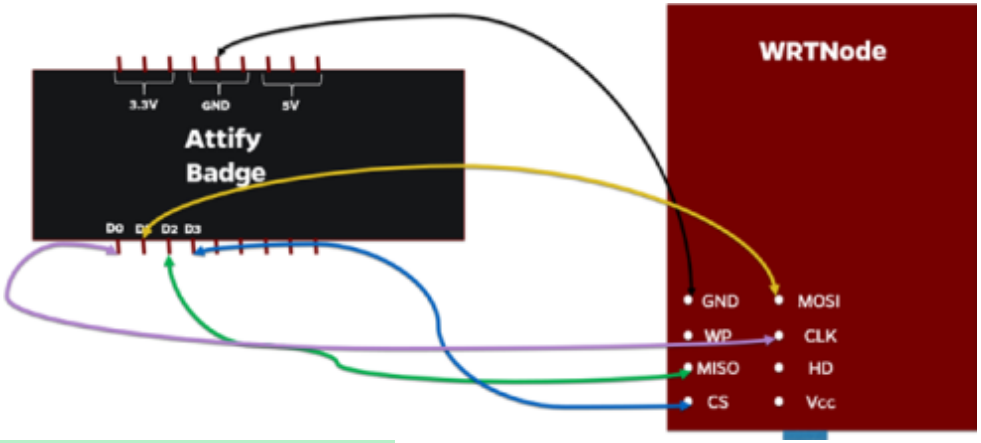


::.. SPI Exploitation (6/6)

WRTNode pin	ATTIFY BADGE
GND	GND
SPI_WP	-
SPI_MISO	D2
SPI_CS	D3 (CS)
SPI_MOSI	D1
SPI_CLK	D0 (CLK)
HD	-
3.3VDD	-

The pins of WRTNode and Attify Badge must be properly connected to support a communication by means of the SPI protocol.

After the connection, the next step is running spiflash.py by specifying a large enough size so that the entire flash chip content gets dumped.



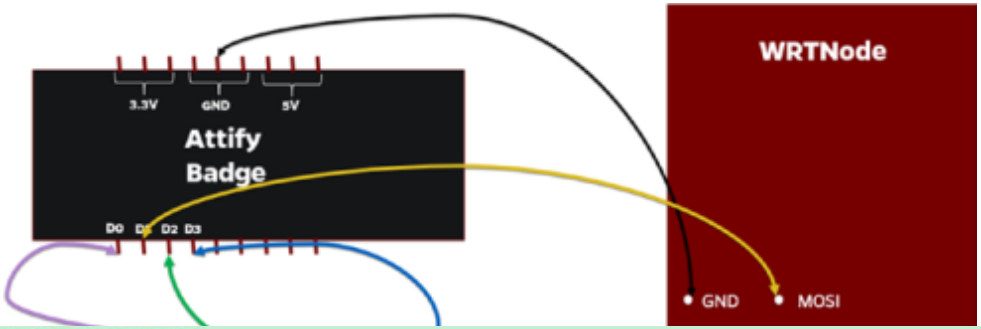

```
→ examples git:(master) ✗ sudo python spiflash.py -r wrtnode-dump.bin -s 20000000
FT232H Future Technology Devices International, Ltd initialized at 15000000 hertz
Reading 20000000 bytes starting at address 0x0...saved to wrtnode-dump.bin.
```

::.. SPI Exploitation (6/6)

WRTNode pin	ATTIFY BADGE
GND	GND
SPI_WP	-
SPI_MISO	D2
SPI_CS	D3 (CS)
SPI_MOSI	D1
SPI_CLK	D0 (CLK)
HD	-
3.3VDD	-

The pins of WRTNode and Attify Badge must be properly connected to support a communication by means of the SPI protocol.

After the connection, the next step is running spiflash.py by specifying a large enough size so that the entire flash chip content gets dumped.




```
→ examples git:(master) ✗ sudo python spi
FT232H Future Technology Devices Internati
Reading 20000000 bytes starting at address
```

```
→ examples git:(master) ✗ binwalk wrtnode-dump.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
114816        0x1C000         U-Boot version string, "U-Boot 1.1.3 (Jun 21 2017 - 14:32:01)"
327680        0x50000         uImage header, header size: 64 bytes, header CRC: 0xCA97F83F, created: 2014-08-13 21:00:49, image size: 1029095 bytes
Data Address: 0x80000000, Entry Point: 0x80000000, data CRC: 0x9A4CEAF, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type: lzma
, image name: "MIPS OpenWrt Linux-3.10.44"
327744        0x50040         LZMA compressed data, properties: 0x6D, dictionary size: 8388608 bytes, uncompressed size: 3104924 bytes
1356839       0x14B427        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 7689776 bytes, 1980 inodes, blocksize: 262144
bytes, created: 2014-08-13 21:00:38
9109504       0x8B0000        JFFS2 filesystem, little endian
```

The wrtnode-dump.bin contains all the read data, and can be passed to firmware analysis tools, and get the entire file system.

JTAG and its Exploitation

::.. JTAG (1/10)

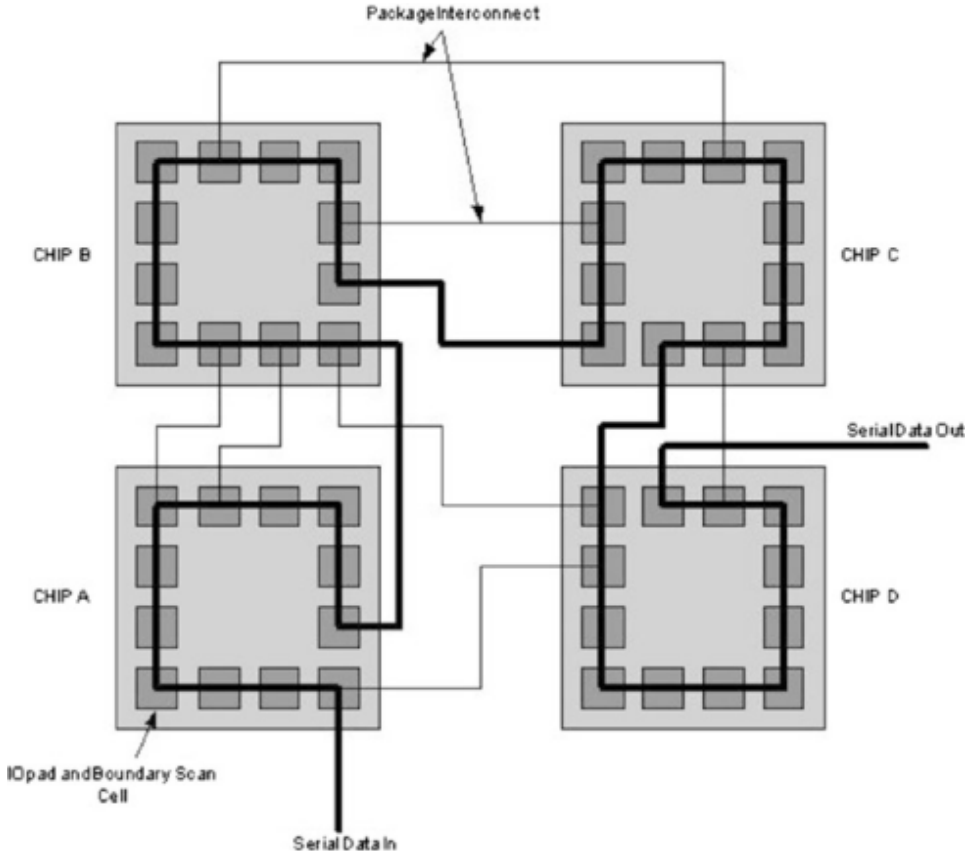
The **Joint Test Action Group** (JTAG) is an association created in the mid-1980s when a group of companies came together to solve the problem of debugging and testing chips while dealing with the increasing complexity of devices.

The manual effort needed to test hundreds of chips with multiple pins in each of them and testing whether each of them is working well and communicating properly started to be overwhelming as the complexity and massive scale production of chips were growing.

To overcome this problem, the manufacturers came up with a standard named IEEE 1149.1 to embed a piece of hardware into the chip itself to allow easier testing.

::.. JTAG (2/10)

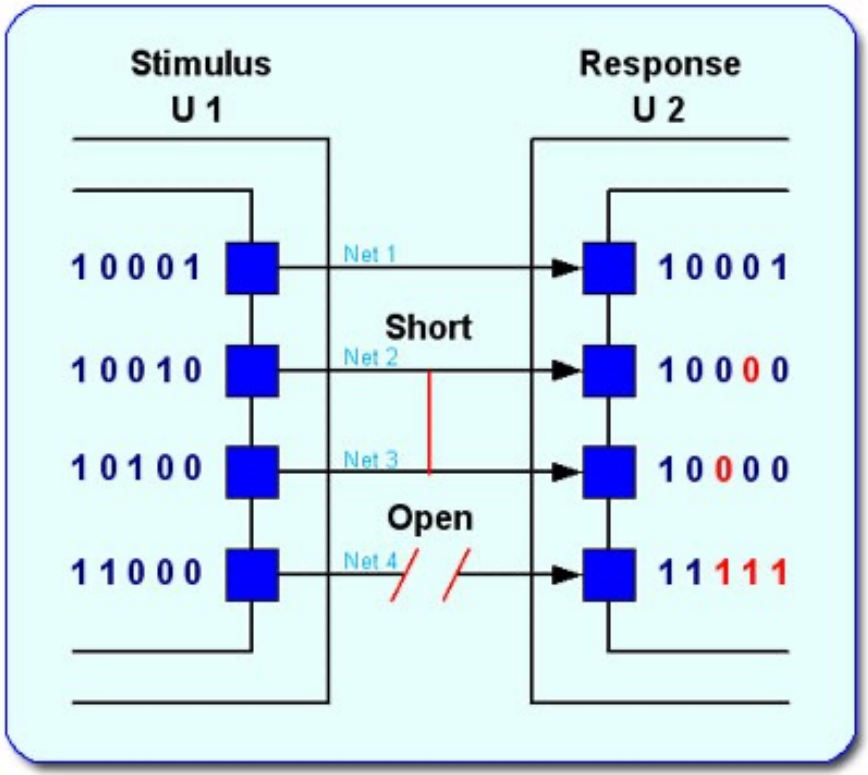
JTAG is just a way of testing different chips present on the device and debugging them by using a technique known as boundary scan. This is done by adding a piece of component called boundary scan cells near each pin of the chip to be tested.



The various I/O pins of the device are connected serially to form a chain. This chain can then be accessed by what is called the test access port (TAP).

The scan happens by sending data into one of the chips and matching the output with the input to verify that everything is functioning properly.

::.. JTAG (3/10)

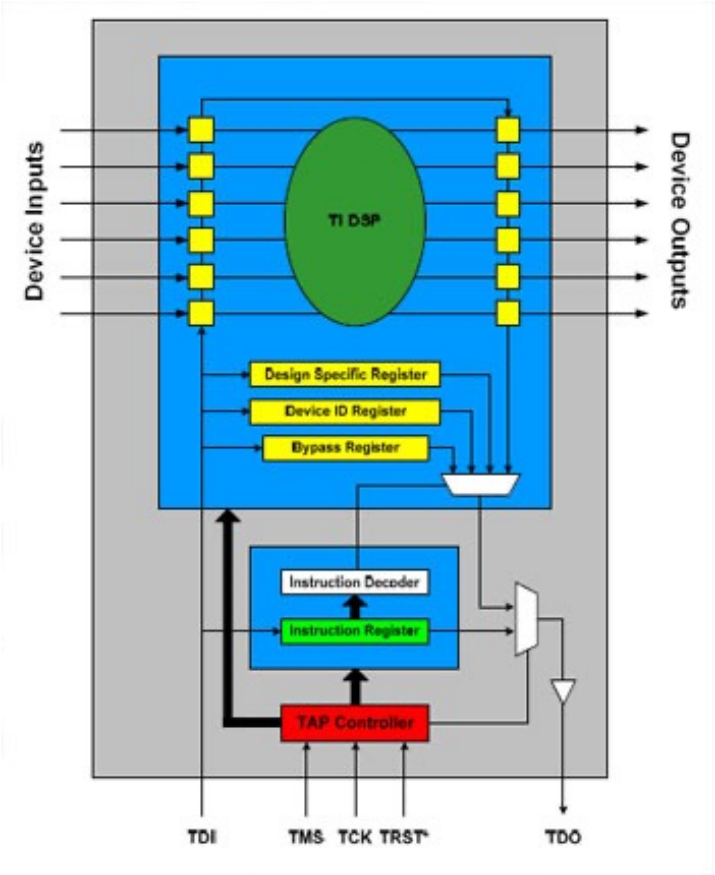


Boundary-scan tool vendors provide various types of stimulus and sophisticated algorithms, not only to detect the failing nets, but also to isolate the faults to specific nets, devices, and pins.

An external file defines the capabilities of any single device's boundary scan logic.

These boundary scan cells can be accessed and checked for the values in the pins associated with them.

::.. JTAG (4/10)

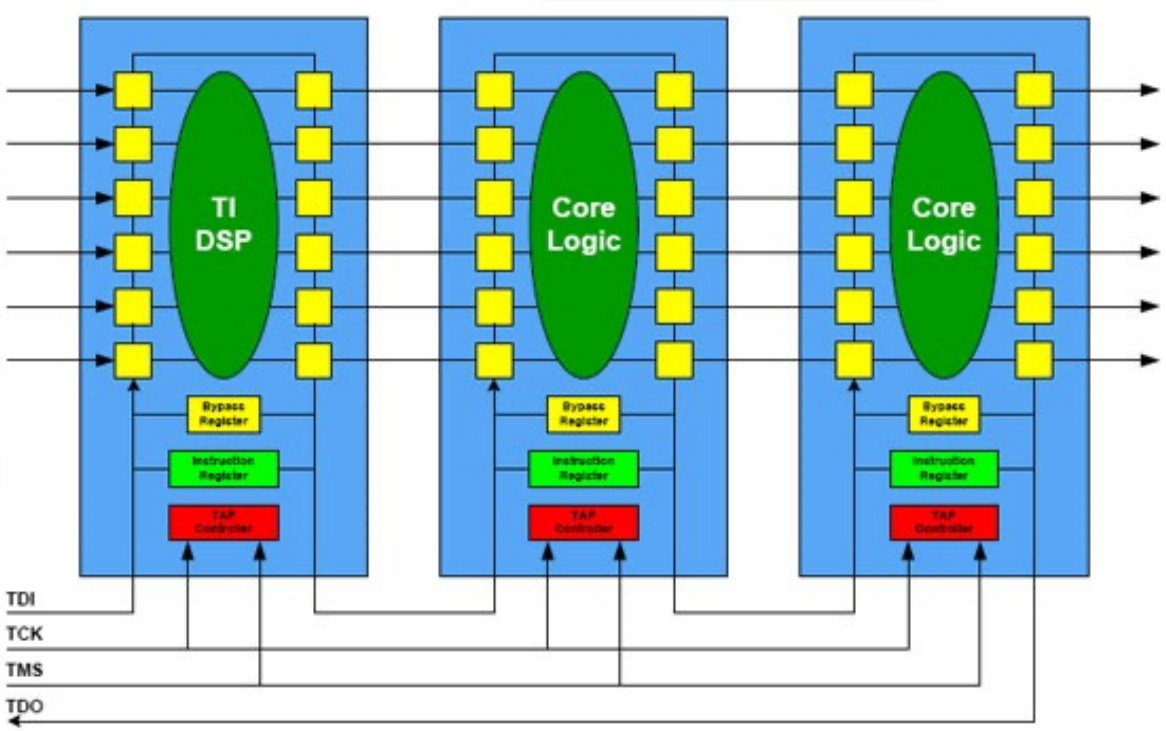


ICs consist of logic cells, or boundary-scan cells, between the system logic and the signal pins or balls that connect the IC to the PCB. Each cell provides specific test capabilities.

The boundary-scan cells within a device are connected together to form a shift register, which is accessed through a serial test data input (TDI) and test data output (TDO) interface.

A standard controller is defined that enables a minimum set of functions to be performed, such as scanning data in or out.

::.. JTAG (5/10)



The mechanism can be used to test a given device internals or even the connection among devices.

Boundary-scan test software can utilize one component to drive signals that will be sensed on a second component, verifying continuity from pin-to-pin.

Devices can be placed in BYPASS mode to shorten the overall length of the chain to reduce test time.

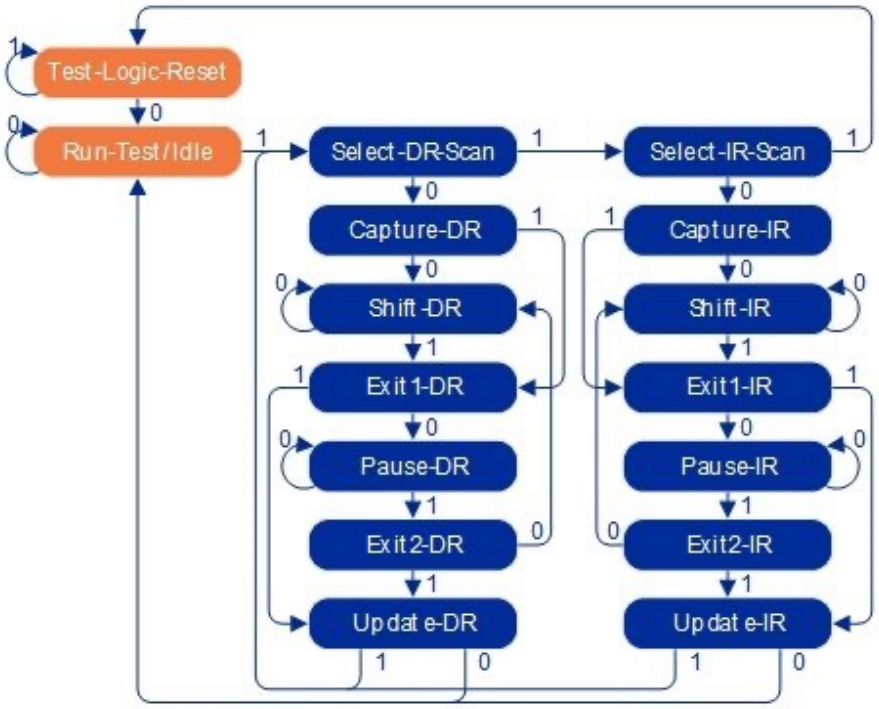
::.. JTAG (6/10)

TAP is a collective name given to the JTAG interfaces present on a device. There are five signals used by TAP :

- Test clock (TCK): To synchronize the internal operations and to clock serial data into the various boundary cells.
- Test data in (TDI): The serial input data pin to the scan cells.
- Test data out (TDO): Sends the data from the scan cells.
- Test mode select (TMS): To control the state of the controller.
- Test reset (TRST, optional): The reset pin that is active low. When it is driven low, it will reset the internal state machine.

The TCK, TMS, and TRST pins drive the 16-bit TAP controller machine that manages the overall exchange of data and instructions.

::.. JTAG (7/10)



The TAP controller is a 16-stage finite state machine (FSM) that proceeds from state to state, based on the TMS and TCK signals. It controls the test data register and the instruction register with the control signals.

If an instruction is to be sent, then the clock (TCK) is activated, and the reset is set to active low for the clock cycle. Once that is done, the reset signal is then deactivated and the TMS is toggled to traverse the state machine for further operation.

::.. JTAG (8/10)

There is set of instructions defined by the IEEE 1149.1 that must be made available for a device in case of a boundary scan:

- The BYPASS instruction places the BYPASS register in the DR chain, so that the path from the TDI and TDO involves only a single flip-flop (shift-resistor). This allows a specific chip to be tested without any overhead or interference from other chips.
- The SAMPLE/PRELOAD instruction places the boundary scan register in the DR chain, and is used to preload the test data into the boundary scan register (BSR). It is also used to copy the chip's I/O value into the data register, which can then be moved out in successive shift-DR states.
- The EXTEST instruction allows the user to test the off-chip circuitry, in addition to SAMPLE/PRELOAD it also drives the value from the data register onto the output pads.

::.. JTAG (9/10)

The overall test process for a boundary scan is run as follows:

- The TAP controller applies test data on the TDI pins.
- The BSR monitors the input to the device and the data are captured by the boundary scan cell.
- The data then go into the device through the TDI pins.
- The data come out of the device through the TDO pins.
- The tester can verify the data on the output pin of the device and confirm if everything is working.

These tests can be used to find problems ranging from a simple manufacturing defect, to missing components in a board, to unconnected pins or incorrect placement of the device, and even device failure conditions.

::.. JTAG (10/10)

JTAG is used to perform several testing and debugging activities.

- Because JTAG is available in the systems from the very start, as soon as the system boots up, it makes it extremely useful for testers and engineers to look at all the various components present in the embedded device.
- For penetration testers and security researchers, it is extremely useful as it allows us to debug the target system and its individual components. If the target board has JTAG access available and contains an onboard flash chip, it is possible to dump the contents from the flash chip via JTAG.

::.. JTAG Exploitation (1/20)

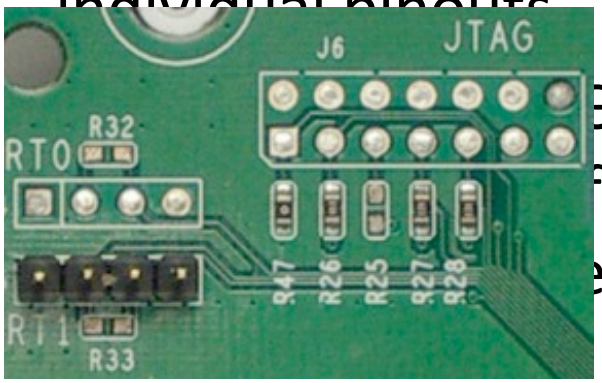
Identifying JTAG pinouts can be a bit trickier compared to what previously seen, where all you would need is to look for a set of three or four pins and then use multimeter to identify the individual pinouts.

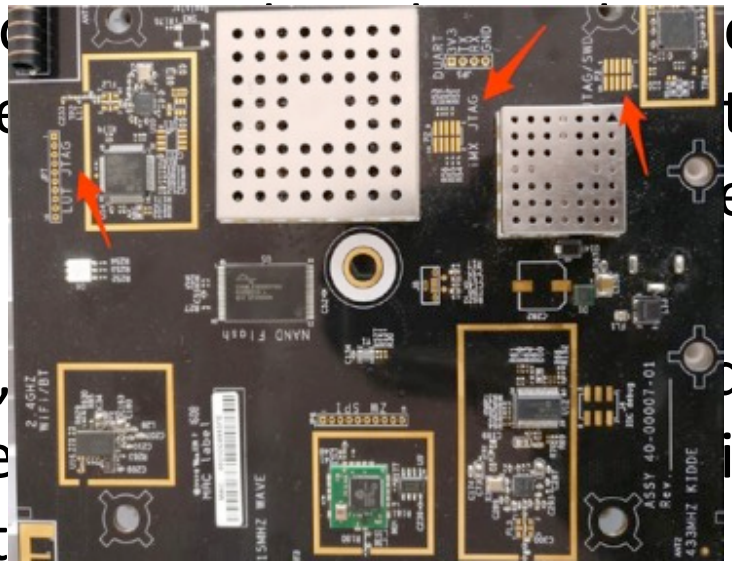
In the case of JTAG, we will need to use additional tools (e.g., JTAGulator) to effectively determine the individual pinouts present in our target device.

Another thing to note while working with JTAG is that in most of the devices you will find the JTAG pads, instead of JTAG pins or pads with holes, which also makes it important for us to have a bit of soldering experience if we want to exploit real-world devices via JTAG.

::.. JTAG Exploitation (1/20)

Identifying JTAG pin compared to what previously seen, whe to look for a set of three or four pins er to identify the individual pinouts



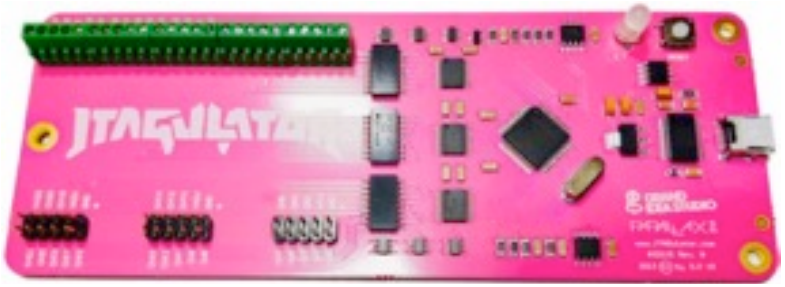


Another thing to note while working with JTAG is that in most of the devices you will find the JTAG pads, instead of JTAG pins or pads with holes, which also makes it important for us to have a bit of soldering experience if we want to exploit real-world devices via JTAG.

::.. JTAG Exploitation (2/20)

We can identify JTAG pinouts using two approaches, which differ based on hardware used:

1. Using JTAGulator, which is an open source hardware, which helps us identify JTAG pinouts for a given target device. It has 24 I/O channels that can be used for pinout discovery and can also be used to detect UART pinouts.



It uses an FT232RL chip that allows it to handle the entire USB protocol on a single chip and enables us to just plug in the device and have it appear as a virtual serial port with which we can then interact using a screen or minicom.

::.. JTAG Exploitation (3/20)

To use JTAGulator, all the various pins on the target device need to be connected to the JTAGulator channels while connecting the ground to ground.

JTAGulator has to be connected to a computer and run a screen with a baud rate of 115200.

```
screen /dev/ttyUSB0 115200
```

On the JTAGulator screen, the next step would be to set the target system voltage by pressing V. After, the next step is to select a BYPASS scan to find the pinouts, by specifying how many channels to be selected for the pinouts.

At the end, JTAGulator will detect the various JTAG pinouts.

::.. JTAG Exploitation (3/20)

To use JTAGulator, all the various pins on the target device need to be connected to the JTAGulator channels while connecting the ground to gro

JTAGulator ha d run a screen with a baud ra

On the JTAGu be to set the target system ext step is to select a BYPAS ing how many channels to be

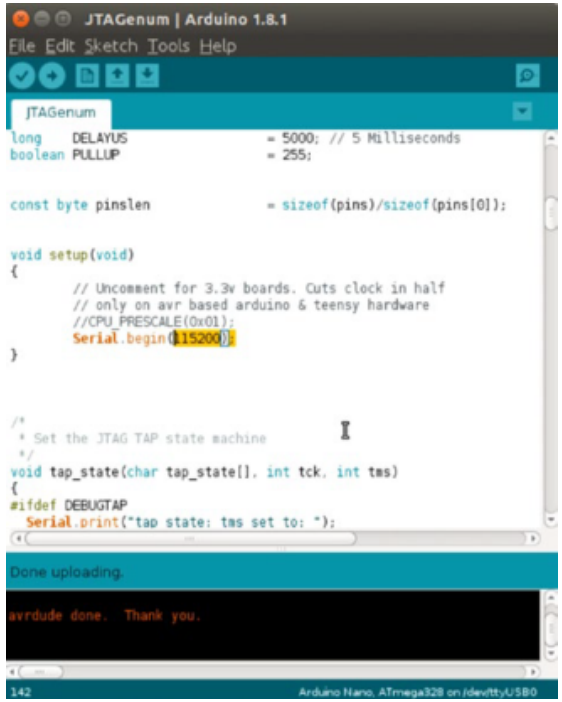
```
:V
Current target I/O voltage: Undefined
Enter new target I/O voltage (1.2 - 3.3, 0 for off): 3.3
New target I/O voltage set: 3.3
Ensure VADJ is NOT connected to target!
:B
Enter number of channels to use (4 - 24): 7
Ensure connections are on CH6..CH0.
Possible permutations: 840
Press spacebar to begin (any other key to abort)...
JTAGulating! Press any key to abort.....................
TDI: 1
TDO: 2
TCK: 6
TMS: 4
TRST#: 0
TRST#: 1
TRST#: 3
TRST#: 5
Number of devices detected: 2
```

At the end, JTAGulator will detect the various JTAG pinouts.

::.. JTAG Exploitation (4/20)

2. Using Arduino flashed with JTAGEnum, a much cheaper alternative compared to JTAGulator, but has few limitations, such as the scan being extremely slow and not having the ability to detect UART pinouts like JTAGulator does.

To use JTAGEnum with Arduino, the first step is to use the JTAGEnum available at https://github.com/cyphunk/JTAGenum.



Once obtained the code sample, the code has to be pasted in the Arduino integrated development environment (IDE). The correct port and Arduino type should be selected from the menu options.

By clicking the Upload button, the code is uploaded.

::.. JTAG Exploitation (5/20)

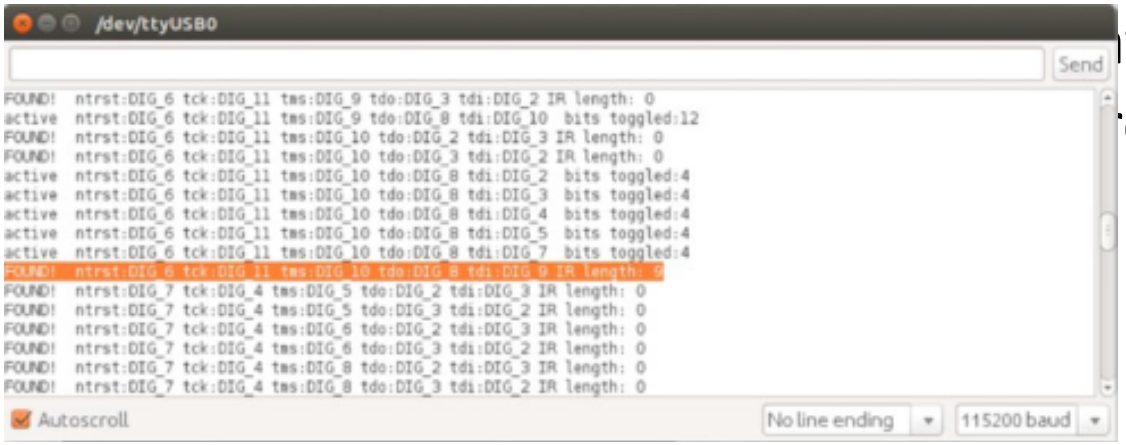
The next step is to interface with the Arduino via a serial connection either through the Serial Monitor present in the Arduino IDE or using a utility such as a screen or minicom.

By pressing s to start scanning, JTAGulator will ultimately tell the JTAG pinouts of the various wires connected to the Arduino.

::.. JTAG Exploitation (5/20)

The next step is to interface with the Arduino via a serial connection either through the Serial Monitor present in the Arduino IDE or using a utility such as a screen or minicom.

By pressing s



tely tell the JTAG pinouts

rduino.

::.. JTAG Exploitation (6/20)

By mapping those wires to the ones connected on the target device, the actual JTAG pinouts are identified. Next, connecting to the JTAG interface will allow debugging the target device and the programs running on it.

OpenOCD helps in such a debug task. In fact, it is an open source software that interfaces with a hardware debugger's JTAG port in order to perform the following actions:

- Debug the various chips present on the device;
- Set breakpoints and analyze registers/stack at a given time;
- Analyze flashes located on the device;
- Program and interact with the flashes;
- Dump firmware and other sensitive information.

::.. JTAG Exploitation (7/20)

OpenOCD has to be installed on the computer by using the apt command or compiling directly the source code available at https://downloads.sourceforge.net/project/openocd/.

An additional useful utility to install would be GDB-Multiarch, which would allow us to use GDB to debug binaries meant for different architectures.

Alternatively, by installing the Attify Badge tool from https://github.com/attify/attify-badge, all the required tools will be automatically installed. The AttifyOS located at https://github.com/adi0x90/attifyos can also be used, where all the required tools are preconfigured.

::.. JTAG Exploitation (8/20)

On the hardware side, JTAG debugging and exploitation can be done with Attify Badge, or other tools such as BusPirate or Segger J-Link, and the target device with the JTAG interface.

To make use of the Attify Badge (or any other hardware for that matter), the OpenOCD configuration file for it and the configuration file for the target device (and for any other devices in the chain).

The Attify Badge will work like a JTAG adapter, and the target device can either be a processor or a controller. To check if our target device's controller is supported by OpenOCD, the target list provided along with the OpenOCD source has to be checked.

::.. JTAG Exploitation (8/20)

On the hardware side, JTAG debugging and exploitation can be done wit sPirate or Segger J-L face.

To make u re for that matter), and the configurat er devices in the cha

The Attify the target device ca eck if our target de the target list provided along with the OpenOCD source has to be checked.

```
~/openocd-0.10.0/tcl/target » ls
1986be1t.cfg                efm32_stlink.cfg      omap3530.cfg
adsp-sc58x.cfg              em357.cfg             omap4430.cfg
aduc702x.cfg                em358.cfg             omap4460.cfg
aducm360.cfg                epc9301.cfg           omap5912.cfg
alphascale_asm9260t.cfg     exynos5250.cfg        omapl138.cfg
altera_fpgasoc.cfg          faux.cfg              or1k.cfg
am335x.cfg                  feroceon.cfg          pic32mx.cfg
am437x.cfg                  fm3.cfg               psoc4.cfg
amdm37x.cfg                 fm4.cfg               psoc5lp.cfg
ar71xx.cfg                  fm4_mb9bf.cfg         pxa255.cfg
armada370.cfg               fm4_s6e2cc.cfg        pxa270.cfg
at32ap7000.cfg              gp326xxxa.cfg         pxa3xx.cfg
at91r40008.cfg              hilscher_netx10.cfg   quark_d20xx.cfg
at91rm9200.cfg              hilscher_netx500.cfg  quark_x10xx.cfg
at91sam3ax_4x.cfg           hilscher_netx50.cfg   readme.txt
at91sam3ax_8x.cfg           icepick.cfg           renesas_s7g2.cfg
at91sam3ax_xx.cfg           imx21.cfg             samsung_s3c2410.cfg
at91sam3nXX.cfg             imx25.cfg             samsung_s3c2440.cfg
at91sam3sXX.cfg             imx27.cfg             samsung_s3c2450.cfg
at91sam3u1c.cfg             imx28.cfg             samsung_s3c4510.cfg
at91sam3u1e.cfg             imx31.cfg             samsung_s3c6410.cfg
at91sam3u2c.cfg             imx35.cfg             sharp_lh79532.cfg
at91sam3u2e.cfg             imx51.cfg             sim3x.cfg
at91sam3u4c.cfg             imx53.cfg             smp8634.cfg
at91sam3u4e.cfg             imx6.cfg              spear3xx.cfg
at91sam3uxx.cfg             imx.cfg               stellaris.cfg
at91sam3XXX.cfg             is5114.cfg            stellaris_icdi.cfg
at91sam4c32x.cfg            ixp42x.cfg            stm32f0x.cfg
at91sam4cXXX.cfg            k1921vk01t.cfg        stm32f0x_stlink.cfg
```

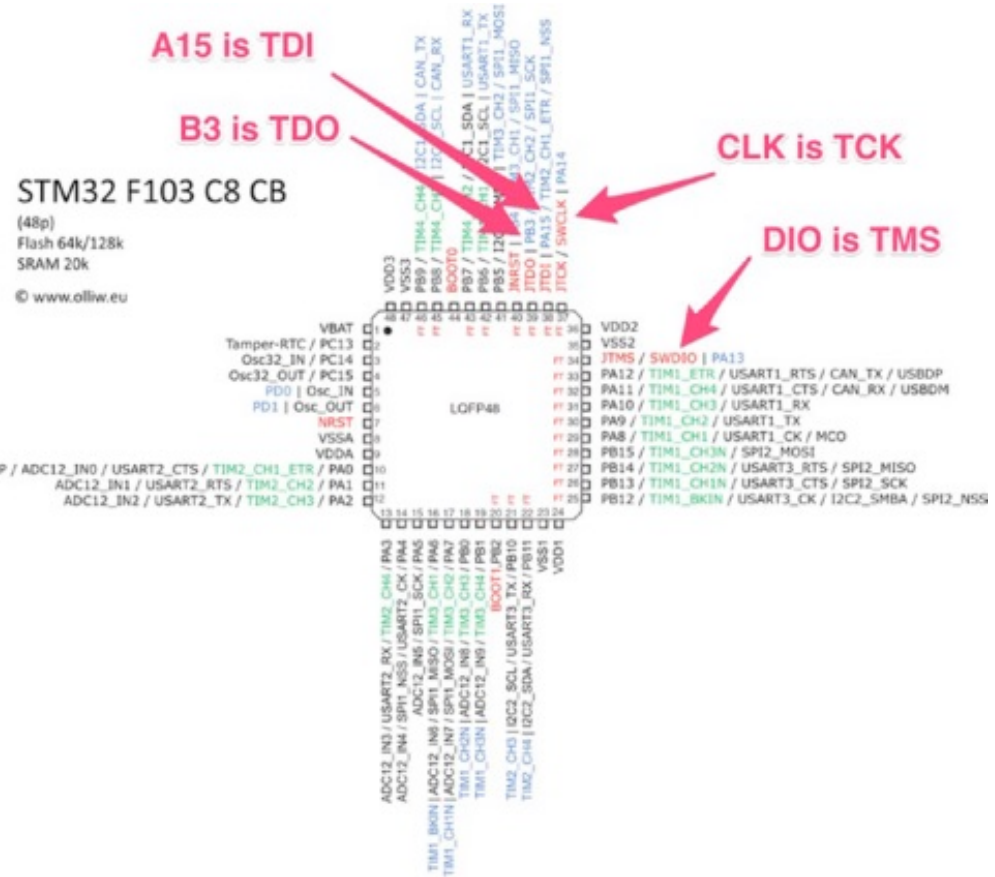
::.. JTAG Exploitation (9/20)

Pin on the Attify Badge	Function
D0	TCK
D1	TDI
D2	TDO
D3	TMS

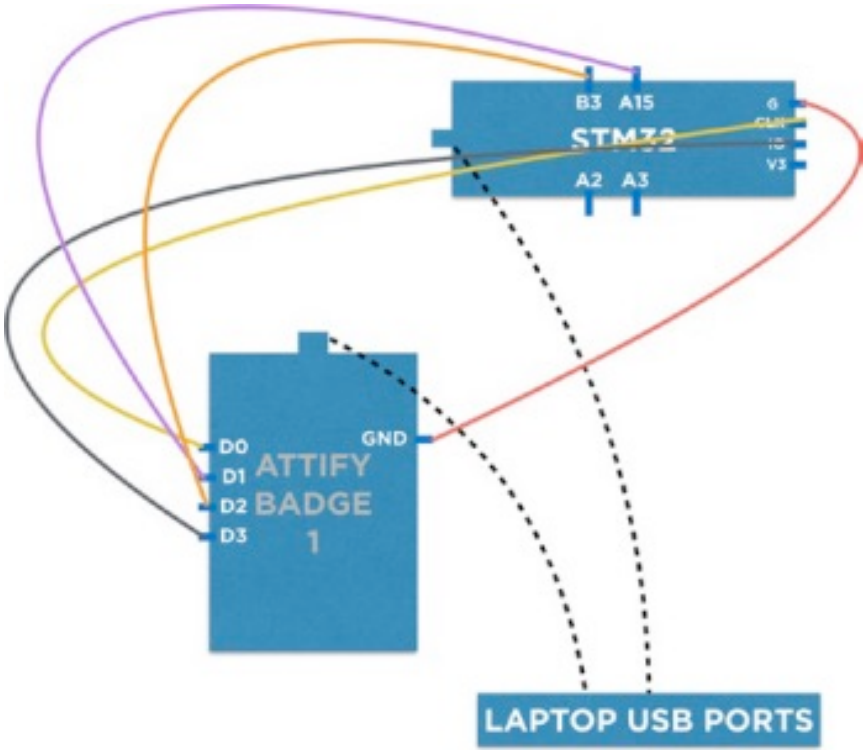
TCK (D0), TDI (D1), TDO (D2) and TMS (D3) go respectively to CLK, TDI, TDO and TMS. The pins functioning as CLK, TDI, TDO, and TMS differ based on the processor or controller of the target device.

A device with the STM32F103C8 microcontroller family has been considered.

The next step is to connect the Attify Badge to the JTAG interface of the target device.



::.. JTAG Exploitation (9/20)



After having connected the devices among each other and with the computer though USB, the configuration files for OpenOCD. The configuration file for the Attify Badge, badge.cfg is as follows:

```
interface ftdi
ftdi_vid_pid 0x0403 0x6014
ftdi_layout_init 0x0c08 0x0f1b
adapter_khz 2000
```

For the configuration file of our target, the STM32 microcontroller, it can be obtained from the OpenOCD configurations itself.

::.. JTAG Exploitation (10/20)

```
$ sudo openocd -f badge.cfg -f stm32fx.cfg
Open On-Chip Debugger 0.7.0 (2013-10-22-17:42)
Licensed under GNU GPL v2
For bug reports, read
       http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
adapter speed: 2000 kHz
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
Warn : target name is deprecated use: 'cortex_m'
DEPRECATED! use 'cortex_m' not 'cortex_m3'
cortex_m3 reset_config sysresetreq
Info : clock speed 1000 kHz
Info : JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
Info : JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

::.. JTAG Exploitation (10/20)

```
$ sudo openocd -f badge.cfg -f stm32fx.cfg
Open On-Chip Debugger 0.7.0 (2013-10-22-17:42)
Licensed under GNU GPL v2
For bug
Info : c
adapter
adapter
adapter_
jtag_nti
Warn : t
DEPRECATED! use 'cortex_
cortex_m3 reset_config sysr
Info : clock speed 1000 kHz
Info : JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
Info : JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
Info : stm32f1x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

OpenOCD is able to connect to the target device and shows additional information, such as six breakpoints, four watchpoints, and more.

Now telnet can be exploited to communicate to the OpenOCD instance, which has connected to the target device over JTAG.

```
$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
```

::.. JTAG Exploitation (11/20)

```
> reset init
JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080009f0 msp: 0x20005000
> halt
```

This implies that the connection to the target device and chip over JTAG using OpenOCD has been successful.

JTAG can be used to write firmware to the device. This is useful when willing to flash a modified version of the firmware to bypass security restrictions on the device. To write a new firmware to the device, the address at which flash starts has to be determined.

```
> flash banks
#0 : stm32f1x.flash (stm32f1x) at 0x08000000, size 0x00000000,
buswidth 0, chipwidth 0
```

::.. JTAG Exploitation (11/20)

```
> reset init
JTAG tap: stm32f1x.cpu tap/device found: 0x3ba00477
(mfg: 0x23b, part: 0xba00, ver: 0x3)
JTAG tap: stm32f1x.bs tap/device found: 0x16410041
(mfg: 0x020, part: 0x6410, ver: 0x1)
target state: halted
target halt
xPSR: 0x010
> halt
```

This implies that the connection to the target device and chip over JTAG using OpenOCD has been

JTAG
wher
bypa
firmw
be determined.

The flash memory in this case starts at the address 0x08000000 and the current size of the contents at that address is 0x0, which indicates that the target device contains no firmware at present. This address can be used in the next command specifying to write a custom-created firmware, firmware.bin.

```
> flash banks
#0 : stm32f1x.flash (stm32f1x) at 0x08000000, size 0x00000000,
buswidth 0, chipwidth 0
```

::.. JTAG Exploitation (12/20)

```
> flash write_image erase firmware.bin 0x08000000
auto erase enabled
Info : device id = 0x20036410
Info : flash size = 128kbytes
wrote 65536 bytes from file firmware.bin in 4.109657s
(15.573 KiB/s)
```

The firmware writing completed successfully. This can be verified by performing a flash banks and seeing the change in the storage size of the flash memory.

```
> flash banks
#0 : stm32f1x.flash (stm32f1x) at 0x08000000, size 0x00020000,
buswidth 0, chipwidth 0
```

JTAG with the dump_image command can be used to dump the firmware from the file system.

```
> dump_image dump.bin 0x08000000 0x00020000
dumped 131072 bytes in 1.839897s (69.569 KiB/s)
```

::.. JTAG Exploitation (13/20)

JTAG can be used to selectively read data from specific memory addresses. This is useful when the exact address that we want to read is known and later maybe modify such data.

```
> mdw
mdw ['phys'] address [count]
 stm32f1x.cpu mdw address [count]
in procedure 'mdw'
```

The command mdw followed by the address and the number of blocks to read does it.

Let's consider a firmware containing an authentication function for UART access, with the password stored at an offset of d240.

Given the base address 0x08000000, the mdw command dumps the password at the sum of the base with the offset.

```
> mdw 0x0800d240 10
0x0800d240: 69747461 4f007966 6e656666 65766973 546f4920
70784520 74696f6c 6f697461
0x0800d260: 7962206e 74744120
```

::.. JTAG Exploitation (14/20)

Often, it is needed to debug binaries and firmware over JTAG to understand the functionality in a much better way, and to modify some of the register values or instruction sets and change the program execution flow.

Whenever OpenOCD is running for a target to perform JTAG debugging, it also enables two different services:

- telnet over Port 4444, which we use to interact with OpenOCD;
- GDB over Port 3333, which we can use to debug binaries running on the target device.

GDB-Multiarch has to be launched with the binary that we want to debug, which can be the one downloaded from the code samples of the book available at https://attify.com/ihh-download, named authentication.elf.

Once connected, the service must be set arm as reference architecture and point to Port 3333, where OpenOCD has attached the gdbserver.

::.. JTAG Exploitation (15/20)

```
$ gdb-multiarch -q authentication.elf
Reading symbols from Vulnerable-binary-for-gdb.elf...done.
(gdb) set architecture arm
The target architecture is assumed to be arm
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x080009f0 in Reset_Handler ()
(gdb)
```

Now, it is possible to set up breakpoints using either hbreak or the break to better analyze the binary and analyze the entire stack and registers when the breakpoint is hit.

hbreak is used to set a hardware-assisted breakpoint, whereas break can be used to set a normal breakpoint at either an instruction, memory location, or function.

To look at the functions in this binary, the info functions command can be used.

::.. JTAG Exploitation (16/20)

```
(gdb) info functions

Non-debugging symbols:
0x08000000  g_pfnVectors
0x0800010c  deregister_tm_clones
0x0800012c  register_tm_clones
0x08000150  __do_global_dtors_aux
0x08000178  frame_dummy
0x08000218  mbed::Serial::~Serial()
0x08000218  mbed::Serial::~Serial()
0x0800023c  non-virtual thunk to mbed::Serial::~Serial()
0x08000244  non-virtual thunk to mbed::Serial::~Serial()
0x0800024c  doorclose()
0x08000290  dooropen()
0x080002e0  verifypass(char*)
0x08000300  main
0x08000380  mbed::Serial::~Serial()
0x08000392  non-virtual thunk to mbed::Serial::~Serial()
0x08000398  non-virtual thunk to mbed::Serial::~Serial()
0x080003a0  _GLOBAL__sub_I_pc
0x080003e4  __NVIC_SetVector
0x08000424  timer_irq_handler
0x080004f0  HAL_InitTick
0x080005ac  mbed_die
```

The function of interest is the verifypass(char *), to have some insights on this the disassemble command can be used.

::.. JTAG Exploitation (17/20)

```
(gdb) disassemble verifypass(char*)
Dump of assembler code for function _Z10verifypassPc:
   0x080002e0 <+0>:     push   {r3, lr}
   0x080002e2 <+2>:     ldr    r1, [pc, #24]    ; (0x80002fc
                                    <_Z10verifypassPc+28>)
   0x080002e4 <+4>:     bl     0x8003910 <strcmp>
   0x080002e8 <+8>:    cbnz    r0, 0x80002f2 <_Z10verifypassPc+18>
   0x080002ea <+10>:    ldmia.w      sp!, {r3, lr}
   0x080002ee <+14>:    b.w    0x8000290 <_Z8dooropenv>
   0x080002f2 <+18>:    ldmia.w      sp!, {r3, lr}
   0x080002f6 <+22>:    b.w    0x800024c <_Z9doorclosev>
   0x080002fa <+26>:    nop
   0x080002fc <+28>:    bcs.n  0x8000380 <_ZN4mbed6SerialD0Ev>
   0x080002fe <+30>:    lsrs   r0, r0, #32
End of assembler dump.
```

This is a function to compare the user input with the actual password using strcmp at 0x080002e4. If they match, it branches to 0x8000290 for door-open or continues further.

To figure out the actual password, we can set a breakpoint at the strcmp instruction and analyze registers r0 and r1, which will hold the two values being compared. One of these values will be the user input password and the other value will be the actual password.

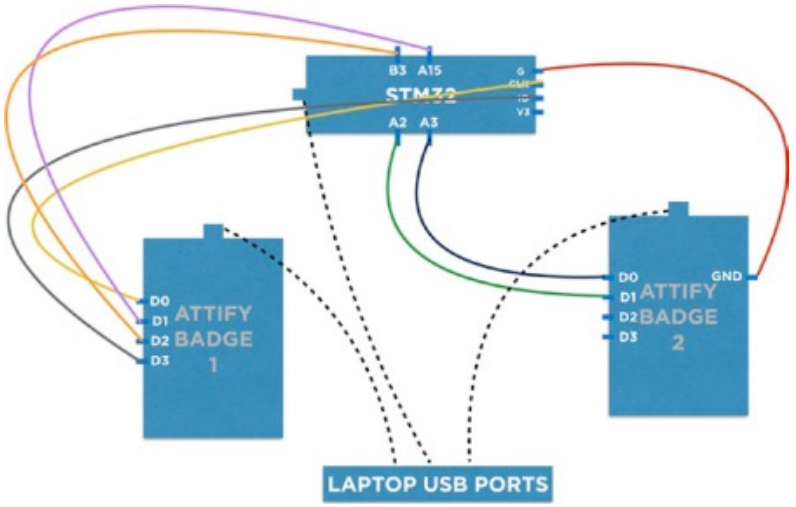
::.. JTAG Exploitation (18/20)

```
(gdb) b *0x080002e4
Breakpoint 1 at 0x80002e4
(gdb) c
Continuing.
Note: automatically using hardware breakpoints for read-only
addresses.
```

Once the breakpoint is set, c can be typed to continue the program execution.

The next thing to do is to connect over UART and provide an input, so that verifypass gets called and our breakpoint is hit. To connect to the same target device over UART, the pins A2 and A3, which is the Tx and Rx of STM32, can be used and connected to D1 and D0 of Attify Badge.

We connect to the UART console in another terminal and use screen to connect to /dev/ttyUSB0 over a baud rate of 9600.



::.. JTAG Exploitation (19/20)

```
$ sudo screen /dev/ttyUSB0 9600
Offensive IoT Exploitation : Enter your Password:
```

After having entered testing as the password, the breakpoint is hit in the GDB session.

//Terminal 1 with UART:

```
Offensive IoT Exploitation by Attify!
                                Enter the password: ********
```

//Terminal 2 with GDB:

```
Breakpoint 1, 0x080002e4 in verifypass(char*) ()
(gdb)
```

::.. JTAG Exploitation (19/20)

```
$ sudo screen /dev/ttyUSB0 9600
Offensive IoT Exploitation : Enter your Password:
```

After having entered testing as the password, the breakpoint is hit in the GDB session.

//Terminal 1 with UART:

```
Offensive IoT Exploitation by Attify!
                                Enter the password: ********
```

//Terminal 2 with GDB:

```
Breakpoint 1, 0x080002e4 in verifypass(char*) ()
(gdb)
```

Now, let's read the registers.

(gdb) info registers

r0	0x20004fe0	536891360
r1	0x800d240	134271552
r2	0x34000000	872415232
r3	0x20004fe8	536891368
r4	0x0	0
r5	0x20004fe0	536891360
r6	0x20004fef	536891375
r7	0x20004fe7	536891367
r8	0xbd7ff3ba	-1115688006
r9	0x9aebfea5	-1695809883
r10	0x1fff5000	536825856
r11	0x0	0
r12	0x20004f50	536891216
sp	0x20004fd8	0x20004fd8
lr	0x800035d	134218589
pc	0x80002e5	0x80002e5 <verifypass(char*)+4>
xpsr	0x61000020	1627389984

::.. JTAG Exploitation (20/20)

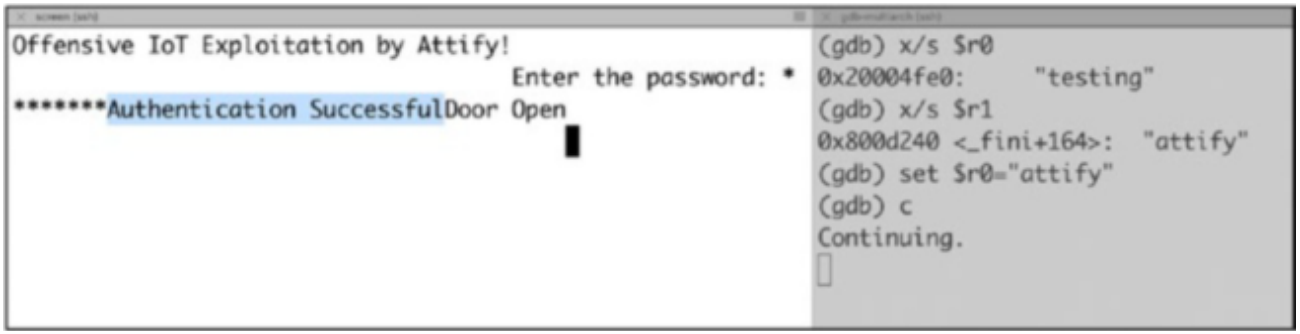
```
(gdb) x/s $r0
0x20004fe0:     "testing"
(gdb) x/s $r1
0x800d240 <_fini+164>:     "attify"
```

The values in r0 and r1 can be seen by using the x/s command, which examines the value as a string.

r0 contains the password entered as input by the user, which is testing, and r1 contains the actual password, which is ‘attify’.

The value of r0 can be changed to be equal to ‘attify’ and type c to continue the execution.

```
(gdb) set $r0="attify"
```



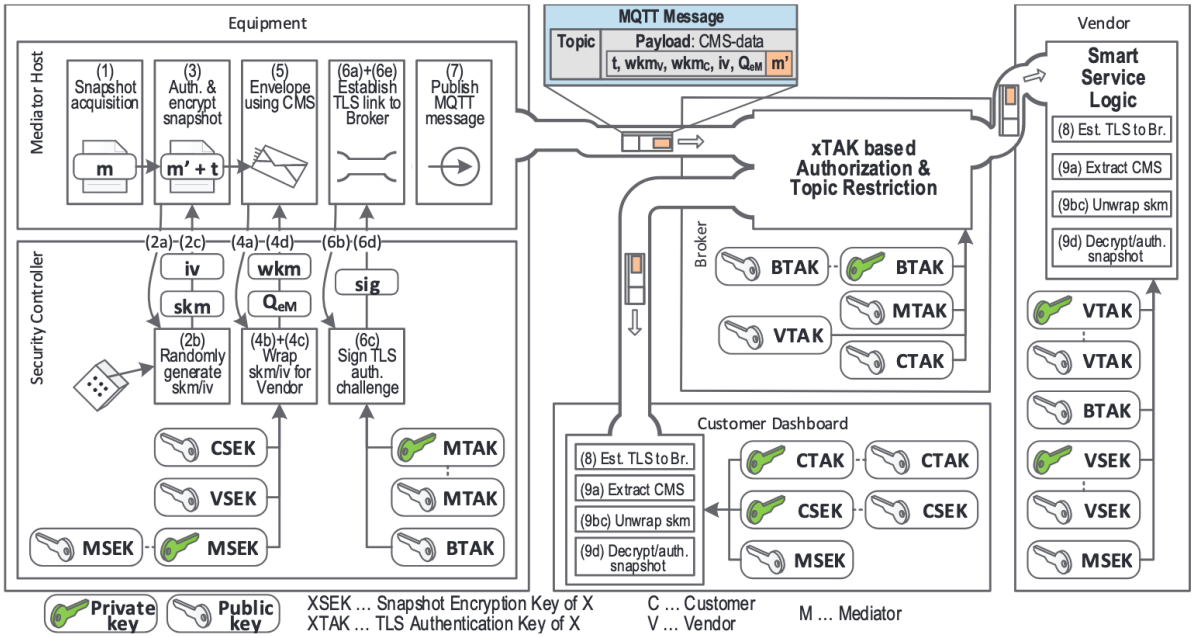
The screen terminal shows that we have been granted authentication.

Protecting Low Level Communications

::.. Serial Comm. Protection (1/2)

A Security Controller (SC) is used in order to provide different communication protocols (e.g. UART, I^2C, SPI,...) with stable but fast communication performance and the independent proof of a high level of security by a security certificate (CC EAL 5+) combined with high cryptographic performance.

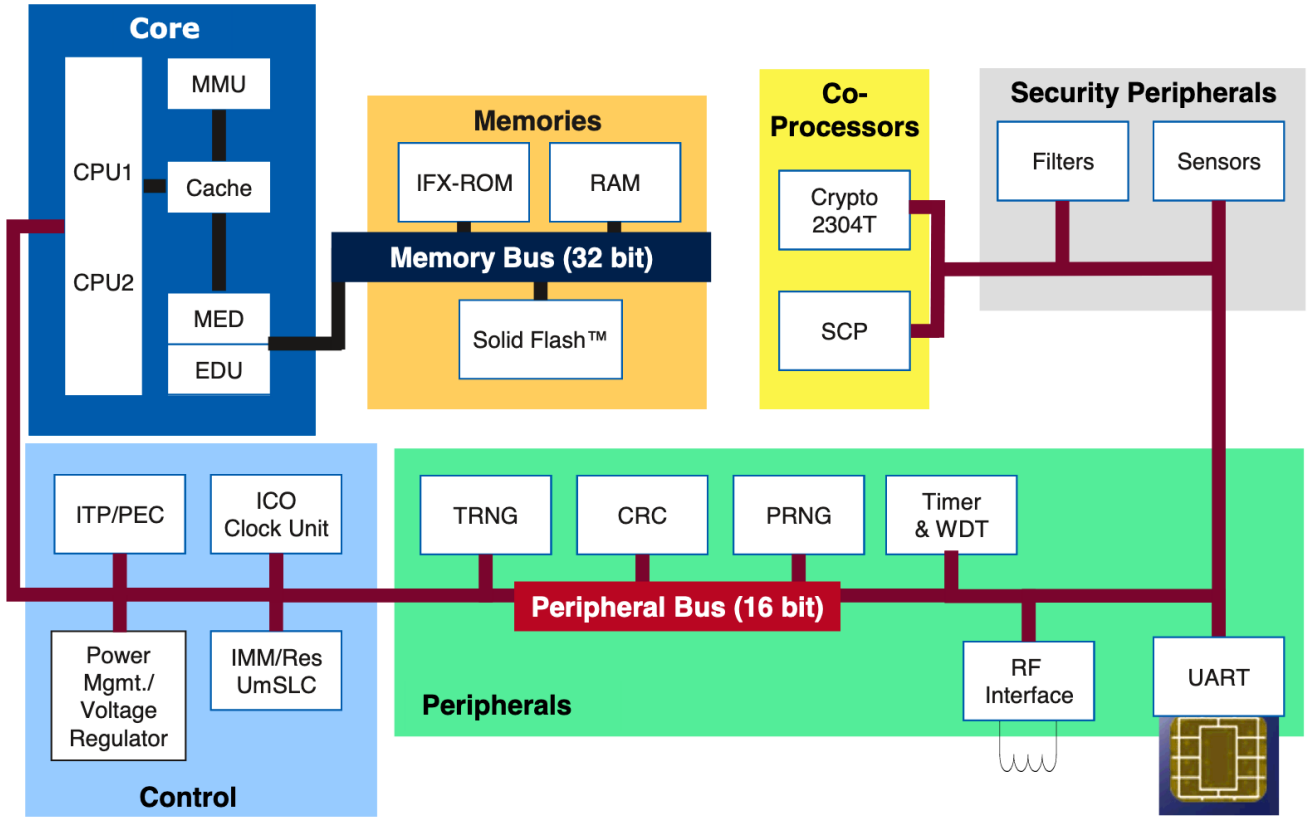
An SC is a discrete hardware module, which can be programmed to provide a defined set of security related functions that operate on the cryptographic credentials stored in the protected storage or generated in hardware.



::.. Serial Comm. Protection (2/2)

An SC provides protection mechanisms against local and physical attacks like probing bus lines. To offer this level of tamper-resistance, SCs can employ within a novel safe and secure IC with dual-CPU concepts, encrypted communication buses, and co-processors.

The internal encryption leaves no plain data anywhere.

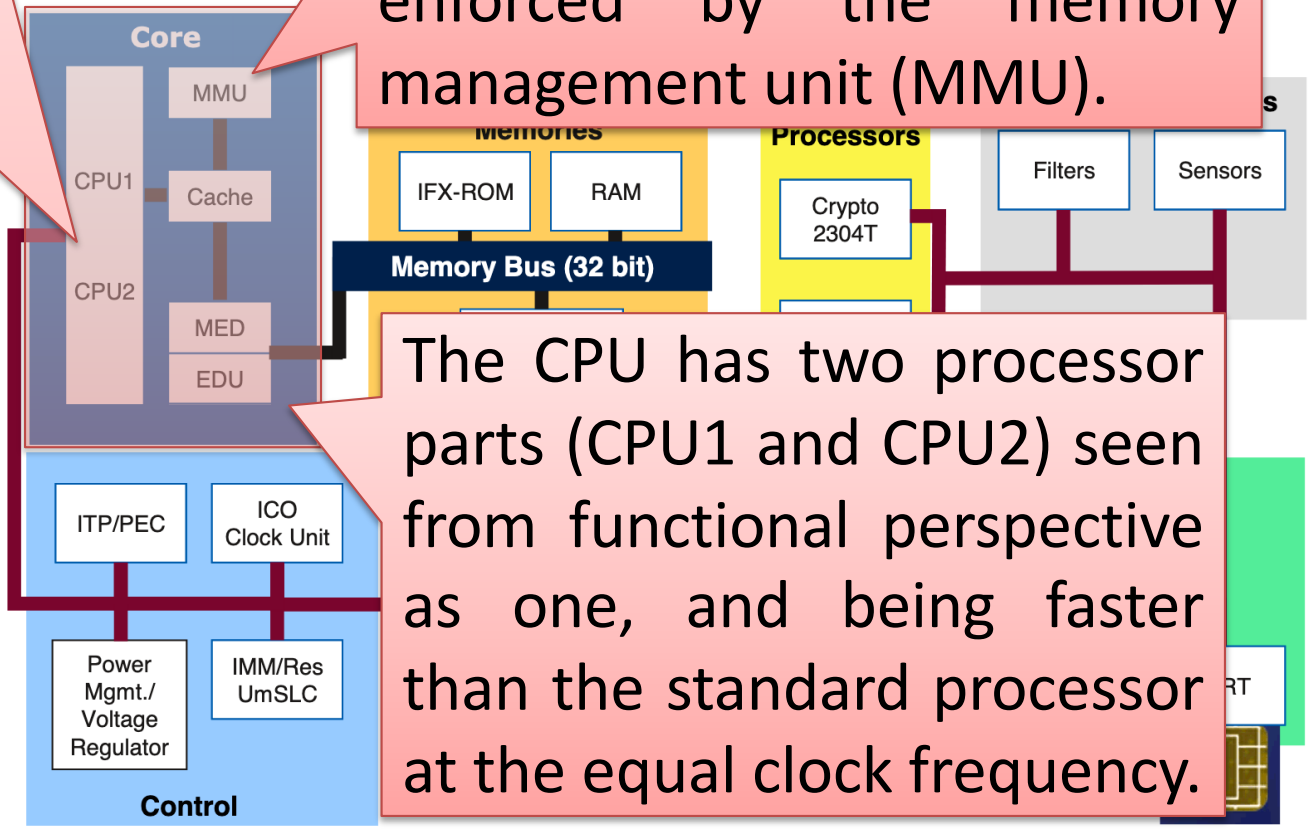


::.. Serial Comm. Protection (2/2)

An SC provides protection mechan
bus lines. T
nploy within
encrypted c

The two processor parts of the CPU control each other in order to detect faults and maintain by this the data integrity. A comparator detects whether a calculation was performed without errors and allows error detection even while processing.

The access rights of the firmware, user operating system and application to the memories are controlled and enforced by the memory management unit (MMU).

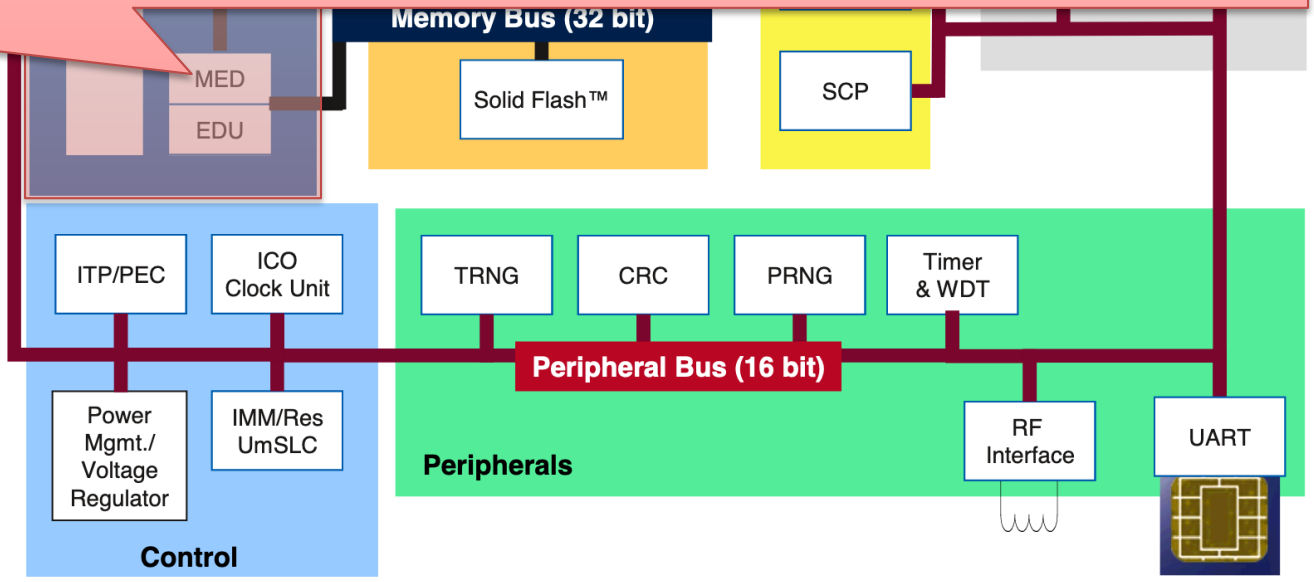


The CPU has two processor parts (CPU1 and CPU2) seen from functional perspective as one, and being faster than the standard processor at the equal clock frequency.

::.. Serial Comm. Protection (2/2)

The CPU accesses the memory via the integrated Memory Encryption and Decryption unit (MED), which transfer the data from the memory encryption schema to the CPU encryption schema without decrypting into intermediate plain data. The error detection unit (EDU) automatically manages the error detection of the individual memories and detects incorrect transfer of data between the memories by means of error code comparison.

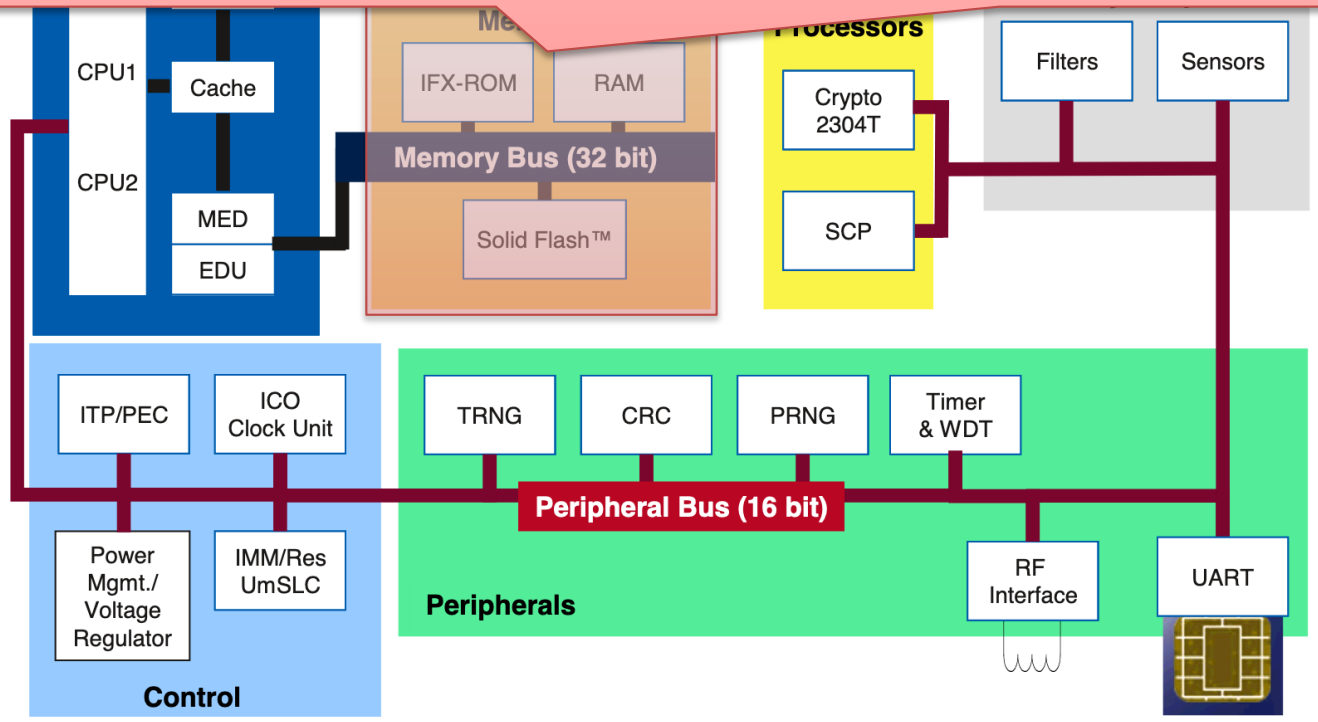
The internal encryption leaves no plain data anywhere.



::.. Serial Comm. Protection (2/2)

The memory block contains the ROM, RAM and the flash NVM. All data of the memory block are encrypted and all memory types are equipped with an error detection code (EDC), while the NVM has in addition an error correction code (ECC). The non-volatile ROM contains the firmware parts and is accessible by the CPU only, while the RAM is a volatile memory and used by the core.

The internal encryption leaves no plain data anywhere.



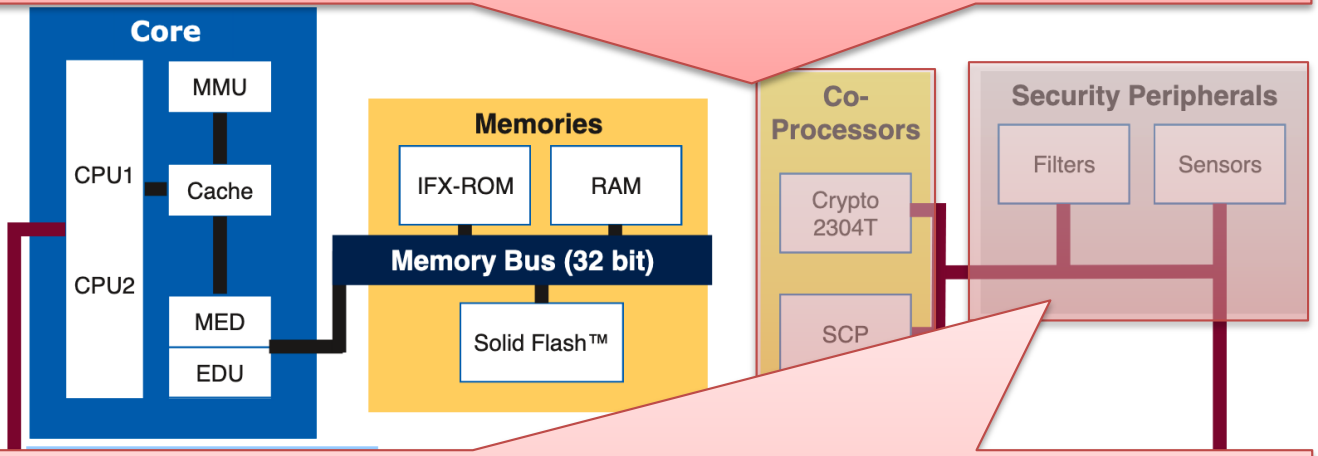
::.. Serial Comm. Protection (2/2)

An SC provides protection mechanisms against local and physical

The coprocessor block contains one co-processor for the calculation of asymmetric algorithms like RSA and Elliptic Curve (EC) and another for dual-key or triple-key triple-DES and AES calculations.

processors.

The internal encryption leaves no plain data anywhere.

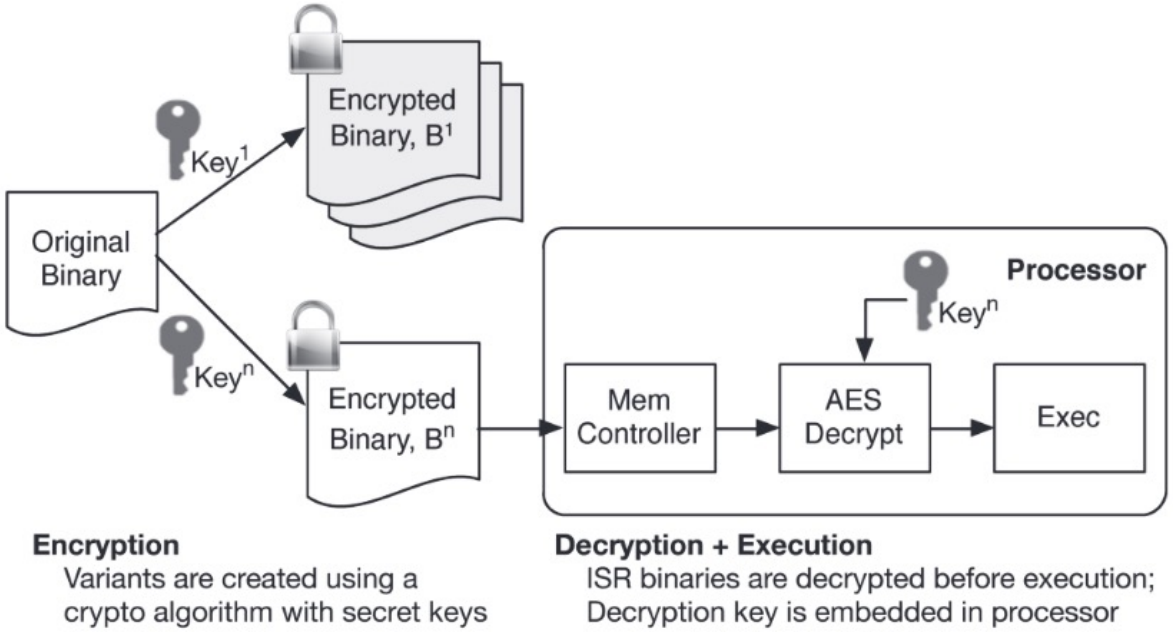


The security peripherals block contains a small set of sensors and filters left in order to detect excessive deviations from the specified operational range, while not being over-sensitive. The small set of sensors is not necessary for the chip security but serve for robustness.

::.. Instruction Set Randomization

Instruction-Set Randomization (ISR) is a general approach for safeguarding systems against any type of code-injection attack by randomly altering the instructions used by a host machine, application, or execution.

For instance, the opcode 0xa may denote the XOR instruction on one application, but may be invalid on a different one. This prevents an attacker from using the same exploit on multiple targets.

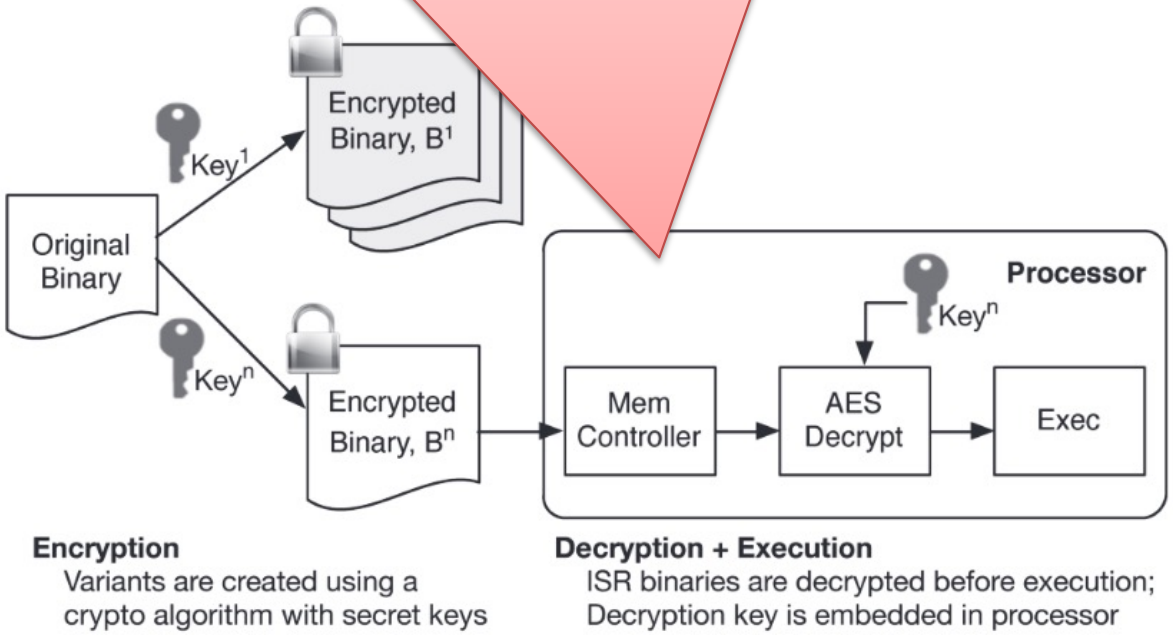


::.. Instruction Set Randomization

Instruction-Set Randomization (ISR) is a general approach for

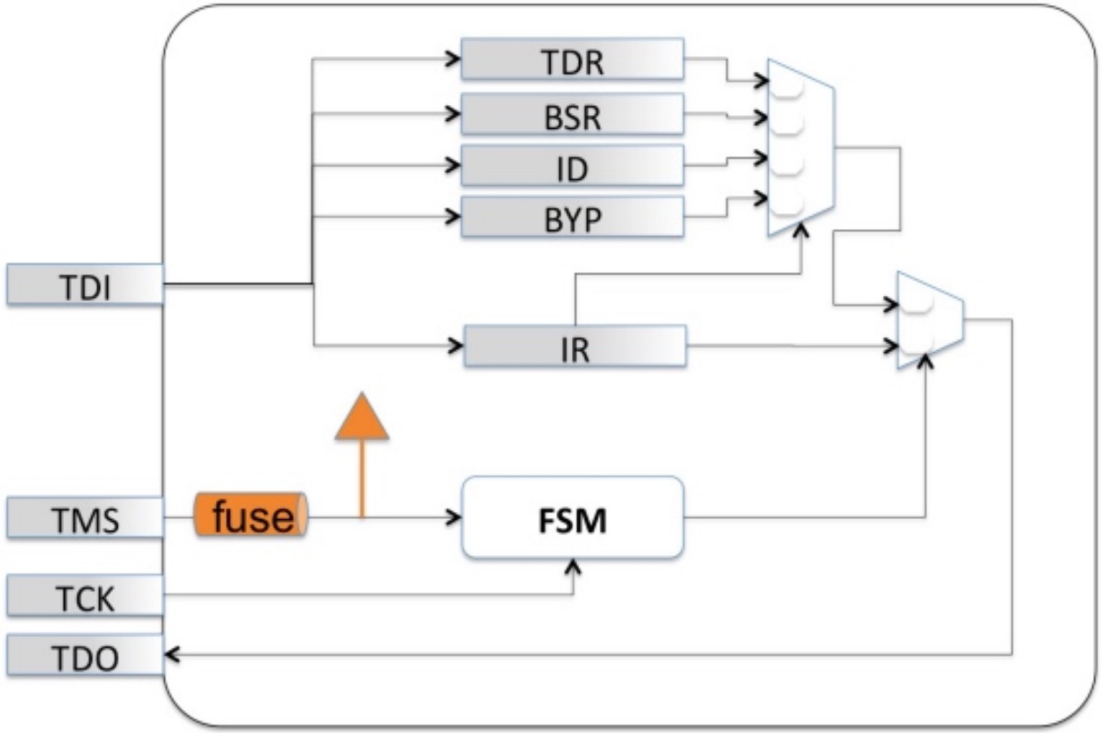
ISR implementations typically "emulate" the random ISA using encryption; code is encrypted at the binary level and decrypted in-memory, before execution.

For instan 0xamay denote the XORinstruction on one application, but may be invalid on a different one. This prevents an attacker from using the same exploit on multiple targets.



::.. JTAG Protection (1/3)

A way to protect a chip from JTAG exploitation is to entirely disable JTAG access. This is often accomplished by fusing off the TMS signal (permanently placing the JTAG finite state-machine in the Test-Logic-Reset state).



JTAG functions are disabled and it is not possible to turn them on again. A better solution is to have a production part naturally be in a “locked” state, and relatively immune to attack, but have some of its instrumentation unlocked in situations where it is critical to have some valuable debug or test engine execute for root-cause analysis.

::.. JTAG Protection (2/3)

A more flexible approach is i.MX6 series System JTAG Controller (SJC) for regulating JTAG access.

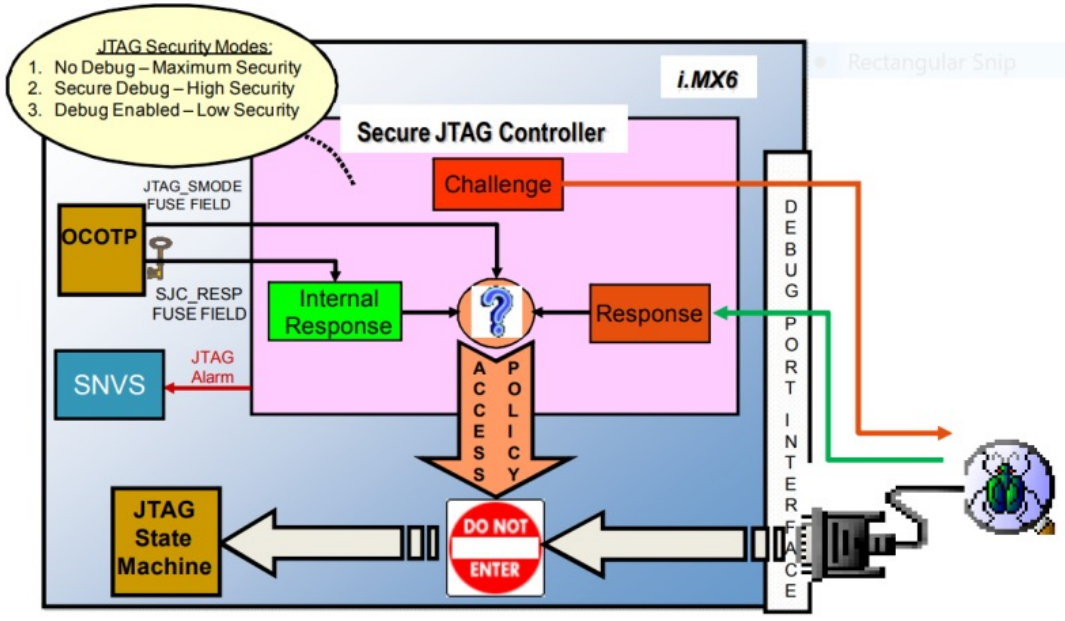
The JTAG security modes are three, and can be configured using One Time Programmable (OTP) eFuses, burned after packaging. In addition to these three modes, there is an option to disable the SJC functionality entirely.

Profile	Description	Test Access	Debug Access
JTAG Disabled	Highest level of JTAG protection. All JTAG features are disabled.	Permanently blocked.	Permanently blocked.
Mode 1: No Debug	Maximum security. All security sensitive JTAG features are permanently blocked.	Always available.	Permanently blocked.
Mode 2 (a): Secure JTAG (without SW enable possible)	High security. JTAG use is regulated by secret key-based authentication mechanism.	Always available.	Available only upon satisfactory response to the invoked challenge.
Mode 2 (b): Secure JTAG (with SW enable possible)	Option for flexibility in Secure JTAG mode. JTAG use is regulated by software-accessible JTAG Debug Enable (DE) bit. Software access to JDE can be blocked until next reset by write-once LOCK bit.	Always available.	Available as above; or on un-blocked software write to HAB_JDE bit.
Mode 3: JTAG Enabled	Low security. JTAG always enabled.	Always available.	Always Available.

::.. JTAG Protection (3/3)

The use of PKI solutions is not viable to the required large amount of computation and latency. A challenge-response mechanism is more efficient and supported by a PUF and a hash.

For authentication ad access control, the SJC submit a challenge to the user, which corresponds to a single correct response. The user messages are not forwarded to the JTAG state machine if the provided response is not the correct one.



::.. JTAG Protection (3/3)

The use of PKI solutions is not viable to the required large
ncy. A challenge-response
orted by a PUF and a hash.
the SJC submit a challenge
ingle correct response. The
the JTAG state machine if
ect one.

A similar method can be implemented internally by utilizing a "locking" segment insertion bit (LSIB) that can only be opened when pre-defined values, corresponding to a key, are present in particular bits in the chain.