Pentesting Hardware - A Practical Handbook

Mark Carney

November 16, 2018

Contents

1 PCB Eye-Spy **13**

1.1 Form Factors 13

1.1.1 Through-the-hole 13

1.1.2 SMC 14

1.2 Basic Components 15

1.2.1 Resistors 15

1.2.2 Capacitors 16

1.2.3 Inductors 17

1.2.4 Transistors 17

1.2.5 Integrated Circuits 18

1.2.6 Other Common Components 19

1.2.7 Miscellaneous 19

1.3 IC Identification 19

1.3.1 SoC 20

1.3.2 Flash ROM 21

1.3.3 RAM Chips 21

1.3.4 Interface Chips 22

1.4 Electricity and Electronics basics 23

2 Toolbox 25
2.1 Hardware Hacking Toolkit 25
2.1.1 Software 26
2.1.2 Connection Software 26
2.1.3 Analysis Software 27
2.1.4 Hardware 30
2.2 Soldering 35
2.2.1 Soldering Basics 35

3 Protocol Overviews 47
3.1 UART 48
3.2 SPI 50
3.3 I^{2}C 51
3.4 JTAG 51
3.5 SWD 54
3.6 Special Mentions 57

4 Things you need to know 59
4.1 Safety First 59
4.1.1 Hardware Safety 59
4.2 Electronics Intelligence Gathering 62
4.2.1 Handy Phrases 62

5 The Hardware Pentesting Methodology 67
5.1 The Hardware Threat Model 68
5.1.1 Motivation 68
5.1.2 What hardware hacking means 69
5.2 Pentesting Hardware - an Overview 71
5.2.1 Hardware Specific Issues 77

5.3 Hardware Pentest Timeline - from Scoping to Reporting . . . 78
5.3.1 Scoping . . . 79
5.3.2 Testing . . . 80
5.3.3 Reporting . . . 82

6 Terminal Access through UART 83
6.1 What we will need . . . 83
6.2 Identifying UART . . . 84
6.2.1 Identify and Connect to Development Headers . . . 84
6.3 Connecting to UART . . . 91
6.3.1 Testing Tx . . . 91
6.3.2 Connecting fully with Tx/Rx . . . 92
6.4 UART Situational Awareness . . . 93

7 Fimware Dumping through SPI 95
7.1 What we will need . . . 95
7.2 Identifying SPI . . . 96
7.2.1 Standard Analysis . . . 96
7.2.2 What to do when Standard Analysis fails . . . 97
7.3 Dumping firmware with SPI . . . 98
7.3.1 Firmware Dumping through UART and `tftp` 100

8 Firmware Analysis 103
8.1 Extracting Firmware . . . 103
8.2 Trophies and Loot . . . 104
8.3 Correlating with UART . . . 107
8.4 Reverse Engineering Firmware and Binaries . . . 108
8.4.1 Reverse Engineering ARM/MIPS Binaries . 108

8.4.2 Reverse Engineering a Firmware ‘Blob’ . . 109

9 Radio Hacking 111
9.1 Wireless Communications - the Backbone of IoT . 111
9.1.1 Radio Basics 112
9.1.2 Radio Terminology 115
9.2 The Radio Attack Surface 122
9.2.1 An overview of attacks 122
9.3 Attacking Known Radios 124
9.3.1 WiFi Hacking IoT 124
9.3.2 BLE Hacking IoT 125
9.3.3 ZigBee 125
9.3.4 LoRa, LoRaWAN, and SigFox 125
9.4 Attacking an Unknown Radio 125

10 Putting It All Together 127
10.1 Background 127
10.2 Hardware Specific Vulnerabilities 128
10.3 Pentesting Embedded Devices - Technical Overview and Attack Models 129
10.3.1 Web Application 129
10.3.2 Infrastructure Vulnerabilities 133
10.3.3 Wi-Fi and Wireless testing 134
10.3.4 Mobile App/Cloud Integration 134
10.3.5 Denial of Service (DoS) 135
10.3.6 Additional Testing Points 136

11 Virtualizing Firmware 139
11.1 The Problem 139
11.2 The Solutions 140

11.2.1 Emulating Linux-based Router Firmware with Firmadyne 140
11.2.2 Emulating STM32 Firmware with `qemu-stm32` 142

12 Practical JTAG/SWD Debugging 145
12.1 What is JTAG and SWD? 145
12.1.1 Motivation 145
12.1.2 JTAG/SWD Functions 146
12.2 Fully worked JTAG/SWD example 150
12.3 JTAG Disabled Bypass 151
12.3.1 Overview 151
12.3.2 The Problem 152
12.3.3 Disabling Debug 152
12.3.4 Bypassing Disabled Debug 153
12.3.5 Attack Outline 153
12.3.6 Remarks 157

13 Glitch Attacks - Real World Fault Injection 159
13.1 159
13.1.1 159

14 Side-Channel Attacks 161
14.1 161
14.1.1 161

15 IC Decapping 163
15.1 163
15.1.1 163

16 Bootloader Reverse Engineering 165
16.1 165
16.1.1 165

17 Custom Protocol Implementations 167
17.1 How do you solve a problem like a custom protocol? 167
17.1.1 Bit-Banging - a beginner’s guide 168
17.1.2 Basic Example - UART Rx 168
17.2 Implmenting a Custom Protocol w/ Datasheet . . . 171
17.2.1 Analysing the Protocol Wiring 171
17.2.2 Analysing The Protocol Timings 173
17.2.3 Using GPIOs to Implment a Protocol 174
17.3 Reverse Engineering and Implementing a Custom Protocol 179

18 CRP Bypass Techniques 181
18.1 Overview of CRP 181
18.1.1 What is CRP? 181
18.1.2 CRP Bypasses - the Basics 182
18.2 CRP Bypasses 183
18.2.1 UV Light to Reset Security Fuses 183
18.2.2 LDR Register Readout on NRF51822 . . . 188
18.2.3 ‘Heart of Darkness’ Attack 190
18.2.4 Cold Boot Stepping 192

19 Cryptography on IoT/Embedded Systems 197
19.1 197
19.1.1 197

Introduction

Hardware hacking is dope, yo...

Acknowledgements

I am deeply indebted, to ... my directors, producers, make-up artists... whatever.

Copyright

For the current releases, this material is released under Creative Commons v3.0 - quote me all you like, and reference my work no problem, print copies for yourselves, but just leave my name on it.

Part I

Hardware Skillz

First off, we need to talk about what hardware is, so that we can talk about the threats in the next part.

This part will be more preparatory stuff - What software is best to get ahold of? What hardware testing kit is best? How does Protocol X work in a little more detail?

This is not designed to be an ‘undergraduate textbook’, but rather a high-level overview for the interested with links to proper technical descriptions where required.

Chapter 1

PCB Eye-Spy

So, first off, we should discuss what each part of a Printed Circuit Board, or PCB, is - what they look like, what they do, and how they operate.

1.1 Form Factors

There are a few things to note about form factors on PCB's - when researching what a part is, or just working out what it could be.

1.1.1 Through-the-hole

When you get your electronics kits from Maplins/RadioShack/wherever, they are generally through the hole kits - that means that the components poke through the PCB, and you solder on the underside and snip off the extra wire. These are very common for hobbyist kits, but much rarer on production PCB's. As such, I'm going to mention them and move on.

1.1.2 SMC

Surface Mount Components was a bit of a revolution in PCB manufacture. The idea is that you can just make a PCB, put on a few thin blobs of solder paste, use a robot arm to place the components on top of the board, and then bake in an oven until it's all bonded. SMC PCB design was instrumental in the full automation of PCB population and testing. It made life stupendously easier for manufacturers.

SMC's are packaged on reels of plastic that resembles bubble-wrap. Each plastic well contains one component, and the Pick-n-place machines (which, let's be honest, do what they say in the name) can easily load a component into a robot arm and then gently drop it onto the PCB in the precise right place.

Many Integrated Circuits (ICs) are in an SMC format called 'Small Outline Integrated Circuit' or SOIC for short. These allow for easy access to pins with test clips (discussed later on) and allow for easy identification of components and traces between them.

Another common IC package type is Ball Grid Array, or BGA. It allows a chip to be much smaller, but have many more active pins in the smaller packaging, that are resistant to the leg-shortcircuits that can occur when soldering very tiny pins coming from an IC - it is very easy to have short circuits between them, owing to the sticky nature of hot solder on tinned metal surfaces. Indeed, the prevelance of BGA's actually encourages one of my favourite things to find on a PCB - JTAG debug capability.[1]

In the next few sections we'll assume that your PCB is SMC for the most part.

1.2 Basic Components

1.2.1 Resistors

Resistors essentially provide some level of resistance. They are basically essential energy wasters. *Essential??* Oh yes - we need to talk about short circuits for this.

A short circuit is when the power finds a path of such little resistance, that all the energy in our power supply (battery, PSU, perpetual motion machine[2], etc.) will race to the *path of least resistance* - it's not a cliché, it's how electricity flows. So, resistors let us manage this flow by setting different resistances.

The greatest resistance we need consider is through air - hence traces can get very very small and close together, as our voltages are so low, we don't really come into contact with the high energy stuff. Building in a resistor lets us use components without worry of shorting our power supply (lots of fuses and trip switches are in place to prevent massive surges caused by a short circuit - mainly becaues things go 'bang' when that happens).

To measure resistance, use a voltmeter with a resistance measuring setting. I'm not going to talk about colour bands on resistors, as so few of these are ever found in the wild, it's not worth knowing that stuff anymore.[3]

---

[1] See relevant section in the testing chapters.

[2] To any physicists - IT'S A JOKE! Tell your face...

1.2.2 Capacitors

Storage tanks that use an electric field to hold voltage. *...and in normal words?* Put simply, a capacitor is two plates that are very very very close together but don't touch. The various parameters of the body of a capacitor dictate how much energy it can store (measure in Farads, but a Farad is huge, so most capacitors you'll meet are in pico-Farads (pF) or microFarads (μF - that's a Greek 'mu' letter, but very common to see in its place is 'uF'). The electric field between these plates is what stores the energy inside a capacitor.

Capacitors are used for all sorts, but the most common usage is on power lines and in filters - because they store energy, you can discharge that energy when you need some backup. What do I mean? Well, powerlines are notoriously fickle and unreliable. As such, a capacitor can store charge and then 'top up' a power flow to help smooth it out. Most modern electronics are very sensitive to power fluctuations (see the section on glitching), so this is something you'll spot very easily.

These are the things that whistle and whine when a TV power supply is dying, and the larger ones (look like little cylinders) have 3 or 4 arm crosses on top. These are so that when the bit between the plates - the *dielectric*, meaning, 'between the electric bits' - fails (which it can do) you can see it, because this plate will

---

[3] It really used to be a thing, but now with SMC's, it's not really worth our while discussing it.

‘pop-out’ and the component will just look really different from the others. Now you can go open a business fixing TV’s.[4]

1.2.3 Inductors

The ‘other’ electrical energy storage device - these store energy in a magnetic (as opposed to an electric) field. These are commonly used in analogue electronics and signal processing circuits (read: radios such as those in WiFi modules, BLE, etc. etc.) and in power blocks (both as one component of a transformer and in filtering).

They have the ability to block AC current but allow DC current to pass - as such, they are found in switch-mode power supplies - when used in conjuntion with capacitors, you can eliminate lots of noise and ‘ripples’ in the power output.

We’re not going to see a huge amount of these initially, so don’t worry for now, but it’s important to know that they are there, and that they are a thing.

1.2.4 Transistors

Electronic switches. They have three pins - collector, emitter, base. Collector you can consider to be where ’stuff goes in’, emitter is then self explanatory, and the ’base’ is the switch bit. When base is 0V, then nothing flows from collector to emitter. When the base is raised over a particular voltage, then a signal can flow from collecor to emitter. The simplest use is as an amplifier - a weak audio signal is sent to base, and this makes the higher voltage from the collector flow according to the base signal. Digitally, this means that you could replace slow valves in computer circuits - valves being electronic switches that rely on a heating element to give out electrons and are thereby slow and also prone to needing to be replaced.

---

[4] Don’t. There’s a lot of high-energy parts in old TV’s that pack enough of a punch to seriously harm, or even kill you. If you’re daft enough to touch those components, then don’t put yourself on my conscience.

The advent of the transistor was only met by the advent of the LED in the history of electronic engineering - it still may be the most significant breakthrough in this field during the 20th C. regardless.

1.2.5 Integrated Circuits

IC's can be anything. **ANYTHING!** IC's can have moving parts[5], capacitors, transitors, inductors - literally, anything built into them. To be useful, capacitors and resistors and inductors tend to be external (as very small versions don't have the same effectiveness). They're amazing little doo-dads, and we're going to spend most of this book working out what's going on inside them. There are many tricks and tips for getting knowledge about IC's, but in general, this is the whole aim of hacking hardware - accessing what's going on inside IC's that we can't see.

---

[5] For a fun example of why these are important, see the documented issues with SiTime Micro-Electro-Mechanical Systems, or MEMS, clock oscillators, and how being exposed to Helium caused them to stop working... No, really! It was something that only affected iPhones, as they are the only ones to use them. Here's the article on Hackaday: `https://hackaday.com/2018/10/31/helium-can-stop-your-iphone-maybe-other-mems-too/`

1.2.6 Other Common Components

Before we get to some more interesting stuff, let's cover some other common components:

Power Regulators

- these take in one voltage, and either step it up or step it down to another. They're very reliable, and often get very warm (so they could well have a heat sink on them - don't remove it!). They have three pins, generally, and are usuall next to or on a power line.

How to identify a power line? To keep the resistance down, these are often the thickest traces on a PCB.

1.2.7 Miscellaneous

[I'm sure I'll find some misc stuff to put here...]

1.3 IC Identification

So, this section is intended to give you an overview of 'usual packages' that these functioning IC's are found in. That said, there's something that should be noted explicitly:

> The only way to give a certain ID of a chip is via the model number on the top.

Granted, this is not always possible (cf. the 'sandwich' method used to put the RAM on top of the SoC on PiZero boards, or the placement of heat-sinks that take light munitions to remove). Nevertheless, it is a good idea to have an overview of the function of each chip, with some notes about how things are achieved/connected and what to spot.

ProTip: If you can't read the text on an IC, then rub some chalk over it, and lightly brush it off with a cosmetics brush. This will leave the chalk dust in the pits of the etched surface, and will stand out against the black epoxy - this makes is superbly easy to read with no effort.

1.3.1 SoC

The 'System on Chip' was a recent revolution in IC manufacturing. The first SoC was found in a digital watch, and this marked the start of a steady flow of such IC's that would steadily condense greater and greater functionality into smaller and smaller footprints.

The core idea is to have one IC that does all the main funcionality. Nowadays, they generally run a specific firmware loaded off a nearby flash-based ROM chip. They generally do a lot of stuff for the developers these days - they tend to implement USB stacks (saves you big-banging your own), as well as Wi-Fi and Bluetooth/Bluetoot Low Energy (BLE) all in one package with a MIPS/ARM (or some other RISC-like CPU) all built in. They can have their own built in RAM, or they can defer to other chips for this. They can have their own in-built secure storage, to act as a sort of 'bootloader' or just a place to put juicy stuff like private keys and such.

SoC's run the show, and they fetch and load the firmware developed to perform the tasks at hand for a particular device. Thankfully, many many SoC's run firmware that is essentially a pared down linux installation. No GUI, but they can drive screens, interface with protocols and other chips, and crucially, have a standard set of debug/development protocols that you can use to look under the hood of a device's working.

1.3.2 Flash ROM

A ROM (Read Only Memory) chip is effectively a solid state storage chip. Much like you get in USB sticks, phones, SD cards, SSD's, etc. How do they work? Well, they have an array of 'floating gate' transistors that are arranged as if they are in a NAND circuit (a well known layout if you use transistors a lot). A floating gate has a captured electron in an isolated piece of substrate that influences the field effect of the collector. But how does that electron get there? Well, quantum tunneling.[6]

ROM Chips, aside from this modern physics wizardry, are very straightforward devices to look at. They tend to use SPI. (see protocols chapter)

1.3.3 RAM Chips

Random Access Memory (RAM) is a well known computer concept, but here you don't get RAM on little snap-in boards, but rather long, flat IC's with lots of connections. These tend to stand out as the traces have to be of precisely the same length - as such, serpent-like 'wiggly' traces are common going to/from these chips to the SoC - these keep the trace lengths the same and stop the signals going out of sync.

---

[6] Specifically Fowler-Nordheim, or FM-tunneling

1.3.4 Interface Chips

USB, Ethernet, Bluetooth/BLE, Wi-Fi - these are all quite specialied protocols, and people long ago coded these into silicon. As such, SoC's will potentially have an interface chip for any protocols they don't natively support, or to act as latency-free support chips (if the on-board system is doing a lot of processing, the on-board ethernet may well suffer, so an external ethernet controller IC may be brought in - very common thing on CPE's).

Common interface-specific IC's you will find include:

- Bluetooth/BLE controllers
- Ethernet controllers - reffered to as 'PHY' IC's, as they correspond to the physical layer of the OSI model
- DECT phone controllers
- Wi-Fi controllers - these are usually shielded on many PCB's so may have to be identified from a bootlog
- USB Controllers - a USB stack takes up valuable firmware space, so for full USB support, small controller chips are used instead

1.4 Electricity and Electronics basics

[This section will be a reference for some basic formulae relating to Electricity and electronics in general. Included as a reference just in case it's ever needed.]

Chapter 2

Toolbox

There's a lot that you need to know about when doing hardware, and some specific skills include:

- soldering
- connecting to hardware interfaces (like UART, SPI, etc.)
- Reverse Engineering PCB's

2.1 Hardware Hacking Toolkit

So, the protocols we will cover in the next chapter are all well and good, but how do you connect to a UART port? How do you talk to a chip over SPI? How do you get your computer to talk JTAG? None of these actions are particularly hard, but you will need some kit.

Here I will present my software and hardware toolkits - I've put these together so that you, the reader, has a ‘shopping list’ of tools and equipment to get/download in one place. Some of this will be repeated in later chapters, along with the protocols in play for a full picture.

2.1.1 Software

2.1.2 Connection Software

First, here's some software that I find useful for connecting to things like a bus pirate (see below):

minicom

A nice interface, reasonably intuitive, however sometimes prone to just not letting you type input to a device. Minicom is very reliable for logging and connecting to devices, and doesn't kick up a fuss nor crash out easily.

screen

‘screen’ is amazing! All your key-shortcuts you had to learn to leave running processes on a server work here (Ctrl+a+d for example). It works as follows:

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

Here, 115200 is the baud rate (see the walkthrough), and /dev/ttyUSB0 is the device you want to connect to. It's really simple, and will also log your output with the -l argument. Swiss-Army tool indeed!

flashrom

Flashrom can be installed using yum or apt-get, and essentially is designed for flashing BIOS ROM IC's. However, it has a config for going over bus-pirate, and is very handy for downloading firmware images (including bootloaders) directly off the flash IC's themselves. Here's an example command:

```
>$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,
    spispeed=1M -o ./firmware-dump.dat
```

This will dump the firmware down in no time! (Well, up to 20 minutes, but hey, there are harder ways of doing it!)

2.1.3 Analysis Software

binwalk

Created by DevTTYS0, binwalk (`https://github.com/devttys0/binwalk`) is a superb firmware analysis tool. Replete with enropy analysis capability, it parses through a binary firmware file and identifies important offsets as well as compression/filesystem types and settings. If somehting is LZMA compressed SquashFS in Little endian mode, binwalk will probably see it and tell you.

The `-e` extract tool is monumentally useful, too - if it sees something it knows, it will parse out that data and decompress it, presenting it to you in a new subfolder for you to inspect. I will be using this in the firmware analysis walkthorughs, so I can thoroughly recommend getting ahold of this.

Jefferson

Where binwalk fails, others pick up the mantle - one such example is Jefferson (`https://github.com/sviehb/jefferson`) Created by sviehb, Jefferson is a JFFS2 compatible firmware analysis tool. This is very useful when analysing firmwares from vendors like Huawei, who have a tendency to use such formats.

'Logic' by Saleae

The Logic package from Saleae (`https://www.saleae.com/downloads`) is a free-to-use logic analysis software. Interfaces with many analysers, and provides a nice suite of on-the-fly analysis tools. Simply set you sample rate top left, connect the logic analyser (see below), press record, then power on the device/signal you want to record.

Analysis includes the ability to play with baud rates to see which work when assessing a UART connection, through to sniffing SPI traffic (if the sample rate is high enough). Supporting Saleae by buying their stuff is worth this software alone - they could charge a lot for it!

OpenOCD

The Open On-Chip Debugger software is an open-source debug protocol suite that interfaces with the latest bus-pirate fimware, as well as many other devices like a GoodFET. To install for use with a bus pirate on a Debian baesd system:

```
sudo apt-get install libtool autoconf texinfo
    libusb-dev libftdi-dev
git clone git://git.code.sf.net/p/openocd/code
cd code
./bootstrap
./configure --enable-maintainer-mode --disable-
    werror --enable-buspirate
make
sudo make install
```

You'll then need to add a `MyConfig.cfg` file with the following contents:

```
source [find interface/buspirate.cfg]

buspirate_vreg 0
buspirate_mode open-drain
buspirate_pullup 1

buspirate_port /dev/ttyUSB0
```

Alternatively, if you're running with a Shikra or other FT232H board, you'll need a config like this:

```
interface ftdi
ftdi_vid_pid 0x0403 0x6014
ftdi_layout_init 0x0c08 0x0f1b
adapter_khz 1000
```

In either case, copy this file to `/usr/local/share/openocd/scripts/interface`, or wherever your openOCD installation locates its script files. You can then run OpenOCD with the following command:

```
sudo openocd -f interface/MyConfig.cfg -f target/<
    target-device>.cfg
```

In this command, `<target-device>` refers to the config file for your target. We'll cover more detail on how these files work in the JTAG chapter.

You can now access the OpenOCD running instance by going to `telnet localhost 4444` and interacting with the device.

Using GDB will be discussed in the advanced section, but once started, GDB can connect once loaded by typing

```
(gdb) target remote localhost:3333
```

We'll cover much more detail later on about how JTAG/SWD works and how to effectively use them for debugging hardware. This quickstart guide, however, should be enough to get you going if you are having some issues.

2.1.4 Hardware

Arduino (any flavour)

The Arduino broke free the microcontroller (MCU) world to hobbyists and hackers alike! [I will likely do a full section on this later] From implementing a custom protocol, to using them for general purpose hackery, they are very versatile and useful devices. Having a few of them hanging around has saved my life on more than one occasion; my most common use for them is bit-banging[1] some protocol or other in order to get the target IC to believe that some event/input is taking place.

---

[1] We'll discuss this later, but bit-banging is essentially sending 0's and 1's corresponding to a protocol over a digital output pin by writing binary strings to the pin in order to simulate some functionality.

Bus Pirate

These little beauties are ace! They speak many protocols - the main things to do are as follows. First off, get ahold of a version 3 hardware - the earlier versions are great, but don't support the more advanced firmware.

Second, get ahold of the firmware that supports OpenOCD - Open On-Chip Debugger is a protocol that lets you do debugging in the chip itself by means of the debug protocols like JTAG and SWD. This may seem a little advanced, but it's worth entertaining as an idea - it will give you valuable information, as well as another avenue with which you can dump the firmware.

[I'll put a section here showing how to do the upgrade if it's of interest.]

IC Connectors

There are two main ways that we will connect to IC's.

Spring-Loaded Taps - The first are using spring-loaded 'leg taps' that come attached to 10-line cables, usually with female pin connectors at the other end.[2] These are little connectors with small metal grabbing arms - when you press down, the arms extend and open out. You can then place them over the device pin you want to investigate, and release slowly. The arms will grap onto the IC leg, and you can then access the voltage going across that pin by hooking the other end of the connector to test equipment.

---

[2] I usually refer to them as 'bird's nests', as that's what it feels like you're untangling sometimes.

[Photo of these taps]

8-pin and 16-pin SOIC Clips - SOIC clips are *really* useful, as they are designed for debugging SOIC package IC's. They usually have a marked cable that is supposed to map to the marked leg on the IC. The clip then just sits on top of the IC and lets you analyse the signals from a pinout you connect to the other end of the clip cable (provided with the clip in most cases).

[Photo of SOIC clip and of a SOIC clip connecting to a flash ROM chip]

Both of these connector types are relatively inexpensive, and will serve you very, very well!

UART to USB adapter

Usually marketed as ‘CP2102’ chips, this IC is designed to do USB-to-TTY UART conversion, mapping directly to /dev/ttyUSB0. Connect the device, start screen or minicom, and then start up the target device and you're all set!

Multimeter

These things are simply invaluable! Checking resistances, voltages, continuity - the average multimeter is a versatile and indispensable piece of equipment. A decent one only costs 20, but they last years and will be referenced throughout the book. I will be assuming you have access to one, they're that important.

Logic Analyser

These pieces of equipment are very useful - they are essentially multi-channel digital oscilloscopes. They don't measure fine-grained voltages for tracing the actual voltages on a chip, but so long as it breaks a threshold (usually 0.5V) these will record and send you data that you can then do analysis on. We will cover the precise methodologies for doing this later on, but suffice to say, these are worth every penny!

Saleae (`https://www.saleae.com/`) make some of the best consumer-grade ones around, but many oscilloscopes come with some implementation of a logic analyser built in these days. Saleae's 'Logic' software is also free to download, and compatible with many of the 'imitation' devices found on the market.

Oscilloscope ***(Optional - but only to a point... You will need one sooner or later.)***

Although only really required when doing work with glitching or more advanced testing/reverse engineering, an oscilloscope can be a very handy tool. Essentially, most modern desktop oscilloscopes are self-contained units that allow you to capture and analyse analogue and digital signals on the fly. They have many many options, and proper useage of the most advanced models can yield significant results!

That said, many cheaper devices connect wirelessly or over USB and can be particularly easy to use. Beware, however, as many of the cheaper ones don't isolate their ground signals, as such, creating a ground loop[3] can be a significant risk!!

Bits and Bobs

So, there are things that are just worth having around, the list including:

- Wire - just generic plastic coated wire, and some thin copper enamelled wire is so handy to have. Tin the ends (see soldering) and you've got something you can connect to a PCB in minutes.
- Breadboards - also called 'prototyping boards', these are white plastic boards you can plug into. I'll discuss below how they work and how to use them, but they are a worthwhile investment.
- Spare Component Selections - I recommend having a stock of the following:
  - Resistors - a range between 10Ω and 100kΩ is best.
  - Capacitors - a range of small ceramic ones and larger cylindrical ones is best. From a few pF to a few μF is a good idea. Handy for setting up aux power supplies, if nothing else.
  - Transistors - in case you need them for a specific circuit

---

[3] These occur when a signal has multiple paths to ground - through induction and other actions, they can gather energy and start oscillating. Search online for the kinds of noise/signals that you can generate, and then remember that such powerful signal loops are usually well beyond the tolerances for most IC's.

  - LED's - if only to give a visual signal that something is happening, LED's are low-resistance sources of confirmational light in my circuits.

- Breakout Units - Arduino 'starter kits' come with 1,001 little breakout boards. Sensors, joysticks, stepper motors, LED digital output blocks - there are all sorts in these kits, and they are fairly inexpensive. If nothing, it'll give you something to learn Arduino programming on.

2.2 Soldering

AKA 'How to play around with molten tin' - it's actually a lot of fun. Fundamentally there are three parts to soldering:

1. soldering iron - ideally a soldering station that is temperature controlled, and has a heat-gun for reflow work.

2. solder - leaded or without lead, good quality with a roisin core and at least 2 sizes; thicker 3-4mm and thin 0.5-1mm diameter solder.

3. flux - either a flux pen or flux gel.

2.2.1 Soldering Basics

Solder is an alloy mostly consisting of tin. Tin's melting point is $231.9°C$ [Fix LaTeX] which is very low compared to most metals - it'll melt at the highest temperature of a domestic oven, and as such, is widely used owing to the fact there's quite a bit of it hanging around the planet, and it's relatively safe to use - no special handling is required, melting point is low meaning no high temperature equipment is needed, and it's also quite soft and easy to work with. We use it to seal our copper piping, connect up our copper contacts on PCB's. Tin in the form of solder is a useful material.

THAT SAID - $231.9°C$ will give you a nasty burn, and as always, take great care when soldering. This isn't a toy, it isn't a game, and if you don't think you should put a hot soldering iron there, then you're right. So don't!

Most solder alloys mix tin with silver and copper (and sometimes a fourth element such as manganeses - as in Sn-Ag-Cu-Mn solders used in BGA reflow work) which pulls the melting point down to around $183°C$, or thereabouts, so it can be worth a quick check what the melting point of your particular solder is.

Soldering Technique

Here is a basic solid technique for doing soldering:

NB - I'm assuming you're using a soldering iron tip that is suitable, and solder wire that has the flux built into it. 'Fluxed solder' is much easier to use, so if you're new to all this, I recommend starting with that.

1. Make sure the iron is at full temperature - if that means waiting a while, wait. When you start heating things up, it will pass a lot of energy on to the things you want to work on, so make sure it's good and hot. Got make a coffee or pet a cat or something...

2. ‘tin’ the soldering iron tip - no matter what size soldering tip you are using, dab solder on the end, and wipe it off on a wet sponge. This will leave a shiny surface that solder loves to stick to - *Warning* - it will oxidise and brown very quickly, so this will need to be done regularly.

3. tin all the surfaces you want to solder - tin sticks to tin really well. So, if you’re soldering wires it is best to tin them first. If you’re soldering to a circular contact test pad, then prepare this first (see below for both)

4. Once you’re ready, make sure the PCB is well anchored/secure - if things fall over you’ll damage some component, or even yourself, so avoid leaving things hanging precariously.

5. Once the components are in place and ready to be soldered, use the tip to heat up the area were you want the solder to go, and then push solder-wire onto the hot soldering iron tip

    - What you want in this act is to flow the solder off the iron straight onto the hot tinned surfaces. It’ll flow beautifully once you get the knack of it, and you’ll get perfect solder joints.

Practise makes Permanent!!

This older mantra from my violin teacher at university[4] is the much improved and true-to-life version of ‘practice makes perfect’. ‘Practice makes perfect’ is wrong, if you think about it - if you keep practicing something wrong, then you'll always get it wrong!

---

[4] Yes, I did my UG in Music, and Tony was my violin teacher who drilled this idea into me!

The trick is to do what anyone who gets good at something does - learn a technique, and then practise it slowly, and concentrate on getting the technique down. You'll soon find that your brain takes over in a sub-conscious way, and you'll get the perfect solder joint every time!

To do this with soldering, you can buy 'solder Training Kits' that give you various footprint and some old/obsolete IC's and components to solder to a PCB, and two PCB's to train on. It's worthwhile, and once you have the technique nailed down, it's a fairly straightforward exercise.

Preparing Solder Contact Areas

Here are some tips on preparing wires and contact points for soldering:

- Use an appropriate wire - if the thing you want to solder to is thin/delicate, use a thinner wire
- Strip the wires with plenty of space to get some contact when you join it to a board
- When tinning plastic coated wire:
    - Strip the plastic and make sure the inner wire is clean
    - Twist the wire between finger and thumb to turn the loose copper wires inside hold together in a 'spiral'

  - Place the wire in a holder, with the metal pointing up, and with a tinned soldering iron tip and fluxed solder, cover the end of the wire from the bottom of the stripped section to the top in a smooth quick motion.
  - Allow the wire to cool before touching it - it will only take a few seconds.

- When tinning enamel-coated copper wire:

  - If the wire is 0.2mm thick or less, put a blob of solder on a wedge iron tip, and holding the wire in pliers, dab the end in the blob of solder to remove the enamel and tin the wire, cleaning the iron tip on a sponge thoroughly afterwards.
  - if the wire is 0.2mm-1mm thick the enamel can be removed using a lighter or rubbing the end in extra fine sandpaper. Tin as normal before using.
  - If the wire is >1.0mm thick, it is best to use fine to extra fine sand paper to remove the enamel before tinning the end of the wire.

- Apply solder to contact points you want to solder to:

  - Often a plastic layer is on top of the pad, so use an emory board or scratch the surface with a knife before applying an iron
  - Using a blob of solder on a wedge tip, gently apply the iron to the contact point for around a second, and pull back to see if the solder has adhere.

    - careful repetition of this will prepare the contact pad to be soldered to with a jumper wire

- if soldering to a leg of an IC, line the wire up carefully, and gently rub a tinned iron tip over the pin and wire to bond them. **Be very careful** not to short the pins, and check with a voltmeter to ensure no erroneous continuity (short circuits) have been introduced.

It is also possible to use the ground-plane (the sheet of copper that covers most of a PCB and acts as ground shielding) to anchor some test pins in the following manner:

1. Cut an appropriate number of pins to have an anchor at either end of the jumper (so, if you need 5 pins, cut 7 from the strip of pins)

2. Find a free area of the PCB edge that can accommodate the new pinout, and mark the pins at either end.

3. Scratch off the solder/silk masks from the PCB in order to expose nice shiny orange copper.

4. Solder the pins onto the ground plane:

    - First, apply solder blobs to the scratched-off areas
    - Tin the first and last pins as you would a wire
    - Solder the pins onto the solder contact points we carved out, and check it's secure.

5. Now gently put small blobs of solder onto PCB side of the pins

6. Connect the tinned ends of the jumper wires you want to secure to the pins.

[Pictures to follow]

Doing this after you're identified and soldered jumper wires onto the relevant contact points (usually for a JTAG that doesn't have a development header on the PCB) will produce a 'makeshift' pinout for the interface you want to access.

[More to follow regarding how to solder, soldering tips, and what's going on when you are soldering.]

Preparing BGA for removal and replacement

BGA's are all the rage! It's because of these being used for everything from eMMC to SoC's that we need to have effective means for dealing with these. Thanks to Deral Heiland[5] from Rapid7 for his work on this!

BGA Removal

To achieve this we will need a heat-gun soldering station, some IPA[6] if necessary, and a little patience.

1. First off, clean the area around the BGA chip.
2. Set the heat-gun to an appropriate temperature for the thickness of PCB
    - **Note** - fibreglass, the substrate of a PCB, and copper, the conductive overlay on fibreglass, are both pretty decent conductors of heat.

---

[5] `https://twitter.com/percent_x`

[6] Isopropyl Alcohol

  - As such, it is worth remembering that you need a higher temperature on the heat-gun in order to compensate for this differential.
  - A reasonable guideline temperature is to start at $350°C$ and go up from there if nothing happens after a minute or so.

3. carefully saturate the area around the BGA with heat from the heat-gun, making sure to spread the heat *evenly* around the target BGA.
    - This is important - it's possible to damage the PCB by causing the copper pads to delaminate (*i.e.* pull off from the PCB) or damage the BGA component if you focus all the heat in one area.
4. using tweezers, test the component by nudging it from the side to see if it will move.
5. It it moves, keep the heat present and remove the component.

You can now load this into an eMMC reader or a custom board you made earlier for the purposes of reading the chip off (using, say, pogopins or a read/write clip for the BGA package layout).

BGA replacement

Now suppose that, after all this, you want to put the BGA chip back.[7] This is where Deral's work really comes into play - the following is derived from his talk at Nuit du Hack in Paris, 2018.

---

[7] It's always a good idea to try and disassemble a target device in such a way

For this, you will need:

- A BGA to solder onto a PCB
- An infrared reflow oven (with temperature curves, ideally)
- A vial of solder balls the width of your BGA solder pads
- A microscope
- Some solder removal wick
- A flux pen and IPA for removal of excess flux
- A soldering station with a heat-gun and temperature controlled soldering iron
- Nerves of steel/soothing music[8]

Here's the rough workflow:

1. First of all, use the solder braid and a broad soldering iron tip with a touch of flux to remove excess solder off the BGA footprint on the PCB
2. Apply a thin layer of flux from the flux pen to the surface of the BGA
3. Get the BGA component under the microscope, and using tweezers, carefully place a solder ball on each pad.

---

that you can reassemble it in a working state later on. Many hardware hackers subscribe to this ideal.

[8] If it's your thing, now is the time to break out the Enya.

   - Although this sounds like a task dreamt up as a special for of water-torture, it’s actually a reasonably straightforward process. With a well-anchored steady hand, a reasonably sized BGA can be ’reballed’ in this sense within ˜20mins. No, really.
   - Also worth noting, the flux is going to act as a kind-of adhesive to stop the balls from moving around. Too much, however, and in the next step (putting the chip in the oven) the solder will join together and form a clump. If this happens, you have to clean up, and start again.

4. Carefully move the BGA to the reflow oven, and set the temperature to have a maximum just above the melting point of your solder for ˜1min. On removal from the oven, it should be completely reballed.[9]

5. Apply some flux to the PCB footprint, and carefully place the reballed (and cooled) BGA on the PCB footprint in the proper alignment - there are usually silkscreen symbols to help here.

6. with the heatgun or in the reflow oven - reflow the BGA onto the footprint.

   - for a reflow oven, set the temp to a max of $250^{\circ}C$, and if using a heat gun, pre-warm the area and then set to $350^{\circ}C$

---

[9] This is usually done in factories by a machine that drops a ball of solder onto a pad and zaps it with a laser to melt it on... so this is the long-winded by-hand low-fi DIY version of that.

  - if doing this with with a heat gun - keep an eye out for the BGA ‘dropping’ onto the PCB. As soon as this occurs, you’re done, so STOP APPLYING HEAT!

7. Clean up as usual - with IPA and a sponge/cotton pad for the flux, especially - and do whatever continuity testing you can.

With these methods, you should be able to remove and replace BGA chips with relative ease. There are other techniques, but these ones are reliable and proven to work - though the replacing technique does require some more initial outlay.

Chapter 3

Protocol Overviews

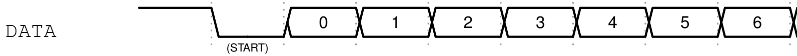
Lots of protocols are in play in hardware, so this chapter will give an overview of the main ones used by electronics IC's/systems to communicate with each other. Ultimately, the aim will be to use these to let us interface with the devices we want to, at a low level. This is the window that hardware hacking opens up to pentesting.

A 'serial' protocol is simply one that acts sequentially, usually in an 'asynchronous' manner (that is, like a turn-based game of chess, rather than a synchronous game of Command & Conquer). We will cover the main ones here, and have some special mentions at the end.

3.1 UART

Universal Asynchronous Receiver/Transmitter, or UART for short, is *so ubiquitous* that of all the many and varied serial protocols out there, we just call it ‘serial’. It’s a very easy protocol to understand - there’s no clock signal (it relies on a preset baud rate), and the only thing to remember is that $Tx \mapsto Rx$ and $Rx \mapsto Tx$ when you wire it up. But first, here’s how the protocol is laid out.

Like many protocols, this is a pull-up voltage protocol. The lines sit at 3.3V until they are pulled down to 0V to indicate a signal. Let’s suppose we want to send a fully btye of data - 8 bits - over the wire from one end to the other. Here’s the layout:



Here is a breakdown of this presentation of the protocol:

- All clock speeds are set independently of the signal - the ‘baud rate’ is the rate at which bits will be transmitted. On this kind of setup, it is just the number of bits per second.
- There is one ‘start’ bit that gets the ball rolling.
- There is one stop bit (a low voltage cycle) that signals the end of the chunk of data
- 8 data bits in the middle
- 1 parity bit at the end of the data

Normally, the settings for (number of data bits/number of parity bits/number of stop bits) is (8/N/1) (**8** data bits, **N**o parity bit, and **1** stop bit). This is generally shortened in screen and minicom to just ‘8N1’.

There are four connections to make this protocol work:

1. VCC - usually 3.3V positive
2. GND - ground
3. Tx - transmit, which is wired to Rx on the other device
4. Rx - receive, which is wired to Tx on the other device

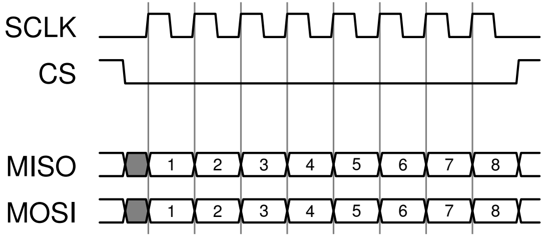
So you can see that there are 2 common pitfalls. The first is wiring the devices up wrong. At worst, this will fry your USB or the device itself. At best, the devices just won’t communicate. **Take care when connecting and powering these pins!** Second main issue is finding the baud rate. Thankfully, you can try many baud rates using the recorded traces from a logic analyser. The most common baud rates are as follows (in order of what I’ve seen the most):

- 115200
- 9600
- 57600
- 19200
- 38400

It’s straightforward to script testing these, but if you try them in this order by hand (when you run screen in BASH for example), you’ll probably find the right one more quickly.

3.2 SPI

Serial Peripheral Interface (SPI) was designed by Motorola to permit the conversation between differnt devices based off a master/slave or controller/source relationship. As opposed to UART, SPI is synchronous, and specialised for high-speed, short distance communication. The timing diagram (due to Martin Scharrer)



Here's what the various parts mean:

- **MISO** - Master In Slave Out - the bus that lets data flow from the slave units to the master unit
- **MOSI** - Master Out Slave In - the bus that lets data flow from the master unit to the slave units
- **SCLK** - The clock signal pin - the rising edge of the clock triggers the level of MISO/MOSI to be read as the current bit by the target device.
- **SS/CS** - in order to select a device, this pin is grounded from 3.3V to 0V, telling the particular device to listen up.
- **VCC/GND** - don't forget that the chip will still need power and ground to operate!

So, as you can see, it's hard to wire SPI connections wrong! MOSI on the target maps to MOSI on the connecting device, and the same for MISO, as well as all the other pins.

NB - for the real geeks who already know this stuff I have chosen the clock phase and polirity to both be CPOL=0 and CPHA=0 in the above diagram.

3.3 I^{2}C

Where one manufacturer goes, another follows. Motorola may have had their SPI protocol, and Philips designed their I^{2}C!

Very unlike SPI, I^{2}C operates as a packet-switched protocol, permitting multiple master units, multiple slave units, and is engineered towards slower, short distance communication.

[Put a diagram here - give a proper description]

Initially, a license fee was associated with using this protocol, but this was waived in 2006. That said, the allocation of ID's to devices is still chargeable.

3.4 JTAG

The Joint Test Action Group (JTAG) had a problem to decide how to test PCB's, especially with the advent of Ball Grid Array and related packages, where the pins are hidden away underneath the package? How do you check if two of the pin connections are bonded with a solder short circuit? What is the best way to solve this problem in a general way?

In 1990, after 5 years of work, the JTAG published IEEE Standard 1149.1-1990, entitled *Standard Test Access Port and Boundary-Scan Architecture*.

Ths solution is roughly as follows - each leg that comes from the main silicon die in the middle of a chip has on its path a small module. These modules are joined up into a ring, and collectively form the testing apparatus for the IC. Given this particular level of access, three main functions were defined for what a 'JTAG' should perform:

1. Debugging - allowing a developer to debug the chip, much like you would debug a local machine with GDB or related
2. Managing Firmware - erasing, uploading, and downloading
3. Boundary Scan Testing - this is testing the Boundary Scan Register (BSR) over the Test Access Port (TAP), or in normal terms, testing the IC pins for problems or issues.

[Put a diagram of how the JTAG looks vis-a-vis chaining chips together, etc]

JTAG doesn't just permit access to an individual chip, but rather, to all the chips on a JTAG chain. This makes debug tasks work more slowly, but gives wider acess to the IC's on the target PCB. Most instances, however, are singular JTAG's for a particular IC.

JTAG is nominally a 5-pin protocol (not counting VCC or GND):

1. **TDI** - Test Data In, this is the input pin.

2. **TDO** - Test Data Out, this is the output pin, and when you daisy-chain IC's with JTAG, the TDO of one goes to the TDI of the next, until it loops back to the debug header.

3. **TCK** - Test Clock, continuing the 'test' naming convention, this is just the JTAG clock signal, the rising edge triggering a read operation. TCK is not chained, but rather forms a 'test clock bus' along with TMS - each IC can see the clock and TMS signals.

4. **TMS** - Test Mode Select, This is read as the clock signal rises, and determines the next state of the internal JTAG controller

5. **TRST** - Test Reset, an optional pin that can reset the internal test controller, but this isn't required.

The internal mechanics are as follows; There is an internal Test Access Port (TAP) Controller, just a mini-chip inside the main chip that manages the JTAG signals the IC receives. There minimally 3 registers - an Instruction Register, and two or more data registers. There is a defined statemachine that uses the TMS level to decide what to do after each clock cycle. This TAP controller connects to the boundary cells (the little bits of silicon I alluded to earlier, that are on every leg coming from the main silicon die) and drives their behaviour. Boundary Cells can raise/lower a leg's voltage to influence the behaviour of the chip.

As such, you can see that JTAG basically controls the chip, and a JTAG unit within an IC, if accessible to a skilled assessor (that's you), can be leveraged to get full control of the CPU stack, access to memory, as well as the ability to dump/upload firmware without pesky restrictions.

[Signal diagram and in depth descriptions to follow]

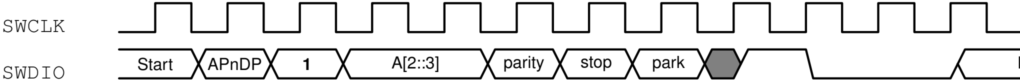
There is also a cut-down version of JTAG called ’reduced pin count’ and this just has TMSC (Test Serial Data) and TCK (Test Clock) pins in play. This, however, was phased out by more accomplished two-wire protocols such as SWD.

3.5 SWD

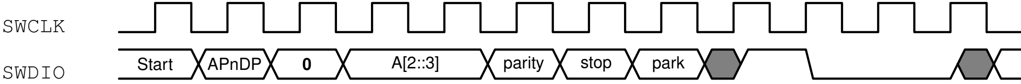
Serial Wire Debug bus was designed by ARM in 2004 to replace JTAG on IC’s that use their processors (so chips that have inside a Cortex MCU, for example). Thankfully, most JTAG modules (like the J-Link by Segger) support this as a two-wire protocol from the data and clock lines (YMMV, so check their documentation).

[PTO for some timing diagrams]

Here’s a timing diagram for a READ operation:


SWCLK
SWDIO
Start
APnDP
1
A[2::3]
parity
stop
park

Here’s a timing diagram for a WRITE operation:


SWCLK
SWDIO
Start
APnDP
0
A[2::3]
parity
stop
park

The differences should be apparent - the ’grey blobs’ are clock cycles where the host shuts up and lets the target talk. The three bits ’100’ that occur after every host opening byte sent is an ACK from the target - if that isn’t received the whole thing loops back to the beginning.

The first byte has the third bit deciding if the READ or WRITE is required, and the other bits decide on the mode (AP (’1’) or DP (’0’) modes) and the ADDR bits decide which register is being talked to/requested from.

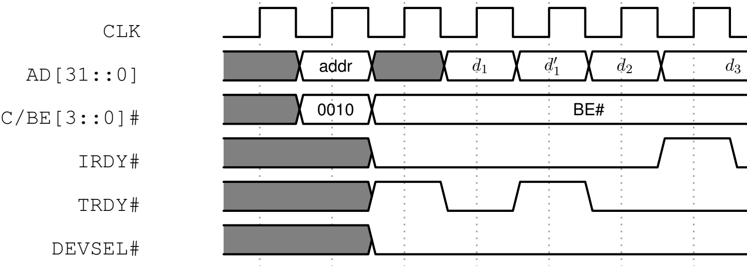
SWD has it’s own little state machine, much like JTAG, and you use the single wire to talk to it, and get access to the debug data (stack traces, etc.) for ARM chipsets. Neat-o.

It is worth noting that for both JTAG and SWD we will explore more examples in the wild in 12.2.

3.6 Special Mentions

Context Area Network (CAN) used in cars. Texas Instruments CC21xx series custom Debug Protocol.

template diagram - for use with ther protocols...:


CLK
AD[31::0]
addr
d_1
d_1'
d_2
d_3
C/BE[3::0]#
0010
BE#
IRDY#
TRDY#
DEVSEL#

Chapter 4

Things you need to know

Before we get stuck in, here are the things you need to know. This is the only chapter you **must** read.

4.1 Safety First

So, some safety pointers from a physical and legal point of view:

4.1.1 Hardware Safety

I remember a handyman at a job I once had in a factory telling me the following maxim for getting stuff done with hardware:

> Right tool for the right job.

That's a very important idea. Although he meant don't use a MIG welder for something a dab of glue would fix, it's a very pervasive notion. Using tools properly and safely is hugely enjoyable! But improper use can lead to some real problems for you.

Here, we'll go through some safety tips - both for your and your person, as well as for the electronics you'll be using.

Soldering

First off, soldering. That thing is hot. Very hot. It's roughly the temperature inside your kitchen oven on its highest setting, except that it's metal and you're holding it in your hand. Soldering irons are very efficient at heating things to this temperature - it's what they're designed for! So, here are some tips when soldering:

1. Use an appropriate rest for the iron when it's not in use.
2. If you're done with the iron, turn it off.
3. Keep children and animals well away.
4. Don't be daft - if you think you shouldn't use an iron on something, then you probably shouldn't.

PCB Handling

You remember at school - you were either shown a video of, or even a real Van de Graaff generator. Someone, usually a girl with long hair, would get on an insulated plinth and put their hand tentatively onto the large metal dome of the generator, and as it whirred her hair would slowly become charged and the strands would float away from each other. Remember that?

Good. The science you saw there was that the human body can hold enough charge as a voltage to literally make your hair stand on end, and not kill you. Sources vary, but it is safe to say that the body can hold hundreds to thousands of volts of charge, and you won't die so long as the current, the number of electrons involved, isn't high. So, as a rule of thumb, volts don't kill you, amps do.

Now think to our poor little IC's. Volts love them, and they love volts. But too many, and they die, never to work again. This is why proper PCB handling is important - it's trivially easy to do the wrong thing, gain a charge, and brush up against a PCB and annihilate all the lovely chips with their secrets.

1. 'Earth' yourself before you touch any elecronics - it's as simple as just touching the exposed copper on a radiator.
2. Use an earth bracelet if you have one - attaching it to the exposed copper of a radiator is a good idea.
3. Socks on carpet is a bad idea.
4. Keep pets at bay, lest they do the job of zapping your target PCB on your behalf.
5. Get an anti-static mat and use it properly - see manufacturer's setup guide.
6. Buy anti-static bags on eBay/amazon and use these to store your previous target PCB's in. **Don't** leave them lying around unprotected in a box if you can avoid it.

4.2 Electronics Intelligence Gathering

Pro tip - being proficient in reading/basic knowledge of Russian/Chinese/German is very useful. Here are a few reasons why -

- Chinese has lots of open manufacturer forums,
- Russia has lots of forums where people have already broken into stuff or gathered information.
- freifunk is a really good resource, along with DeviWiki and OpenWRT wiki/forums.

[Expand on this list]

4.2.1 Handy Phrases

Here are some handy phrases to search for when doing intelligence on IC's on the internet:

German Phrases

Russian Phrases

Chinese Phrases

filename 下载 "Download filename"
数据表下载 "Download the datasheet"

Part II

Basic Assessment Techniques

Murphy’s Law of IoT

> “The router/camera/internet toaster in your home, with access to all your data, was made by the lowest bidder...”

Kirckhoff’s Principle for IoT

> “In any IoT system, some level of security should be assured, even assuming the complete compromise of at least one device.”

Due to Mark C., inspired by Andrew Tierney.

3 Step Plan to IoT Nirvana

We present the following ‘3 Step Guide to IoT Nirvana’[1], originally by Mark C.:

1. Ensure you have per-device secrets.
2. Make firmware updates possible.
3. Make the security problem a named-person’s problem.

---

[1] No guarantee of actual Enlightement is given nor implied by the author.

To include:

- UART locating and connecting walkthrough
- SPI Flash ROM dump walkthrough
- pentesting tips/techniques using UART and dumped firmware.

Chapter 5

The Hardware Pentesting Methodology

To get the most out of security assessing hardware, you should ideally be proficient in numerous areas of security testing - infrastructure, web, mobile apps - as these are all presented attack surfaces on modern devices. Hardware hacking can greatly enhance the testing of these surfaces, and offers up some new attacks and a level of detail that is completely missed if the device's debug/terminal functionality is not explored.

Here, we'll explore the motivations, realistic threats, and give an overview of the testing possibilities we will build on practically in the next chapter.

5.1 The Hardware Threat Model

5.1.1 Motivation

A matter of longevity

The ‘half-life’ of a vulnerability - the length of time for half the users to have their service/software patched against an identified threat - is nowadays very short; minutes for a hosted web service, even hours for a maintained executables and services, to weeks with clients who have strict roll-out testing requirements. However, a vulnerability in hardware will be around for **YEARS**. Once a device is installed, it’s there for the long run, and if there is one guy who procured it, went on the training course, but then left under a cloud and never wrote down how to update it - you’re really stuck until you upgrade. These are the true facts of hardware based security models.

Looking past just getting a shell

There are 1,001 guides online for doing ‘Hardware Hacking’ online, and they mostly stop at ‘...and now you’re connected’, leaving the reader to come up with the relevant attacks, tests, and experience for themselves. Few places deal with what the risks really are, and as has been noted below, the risks as only accumulating. They are not going away.

Yes, I understand (more than most) the value of having a r00t shell on a device, delivered to you through UART, but let’s be honest for a moment - **this is not a viable attack vector!** In the majority of cases, there is no way that you’ll actually execute access to a UART on a victim's device without either their noticing you, or travelling around fully equipped to perform the installation and assessment of the development header.[1]

It is important to fully appreciate what the threat model is from an attacker's perspective, looking at realistic scenarios for attack.

5.1.2 What hardware hacking means

First, a quick digression into what **I**, the author, mean by 'hardware hacking'. This is a very broad topic - it covers circuit bending, repurposing tech, breaking DRM on games consoles, accessing debug functionality on IC's built into product PCB's, designing and getting an arduino/RasPi/(generic microcontroller) to achieve a certain task for a project you're making, accessing/writing to protected ROM areas, reverse engineering IC's that don't have public datasheets, *etc. etc. etc.*

Hardware hacking is a *vast* topic, and I don't want to get bogged down with definitions - they're all valid uses of the term 'hacking'.

Specifically, for me, the phrase hardware hacking is linked to my work as a pentester and security researcher where I would analyse IoT/hardware from the standpoint of an attacker looking to get some malicious leverage on a system - so, my point of view for 'what to do when you have access to a debug/terminal interface with a chip?' is to use this deep level of access to test for vulnerabilities and security flaws. This is what I'm interested in, and so I’m making this explicit from the start.

---

[1] The only examples I have ever found where this is not the case is when such a debug port is publically available, such as Cisco switches/ASA's in car parks without adequate physical security being provided.

As such, here are the kinds of questions I ask of a device I come into contact with:

- What else can you do besides (generic product description)? (you could argue this is a definition of ‘hacking’ in a generic way...)
  - Are you really a microcontroller being used to flash a light?
  - Do you have a firmware-controlled radio that could be repurposed?
  - Do you have some advanced functionality that goes above and beyond what you need for the task at hand, and that I can use for other stuff?
- Can I turn you into a malicious device for blackmail?
  - e.g. bugging, video recording, disclosing sensitive user data or Personally Identifiable Info (PII) - that kind of thing
- Can I get data out of you that relates to your owner?
  - e.g. PII again, usernames, passwords, secret keys for 2FA, private keys for encryption, etc.
- Can I add you to a botnet/BTC miner?
  - easy ways to monetise pwned devices - get them to mine BTC to a wallet you control, or add them to a botnet and sell DDoS services a la Mirai clone botnets

- Can I think of any other malicious activity that would benefit me in some real way, that I can leverage to this device and others?

5.2 Pentesting Hardware - an Overview

We want to get you to understand the attack scenarios and ways in which you can pentest so as to cover off these attack vectors effectively in your work.

So, let us suppose you're a pentester and you've been given a device to hack as part of a test - what now?!? In fact, many categories of attacks relate to Hardware hacking from this security standpoint, in particular:

Outdated Software

There is a major supply chain issue in IoT. A recent mass hack[2] in the UK showed that there are some serious flaws in the generic underlying software presented on the devices. *But how the hell are these vulns here?* Well, in part it's a supply chain problem.

Often (incorrectly) referred to a *ZyXEL*, Allegrosoft[3] manufacture software for embedded systems, including the very popular RomPager server software. It was clear that there was some issue with this software in the wake of this hack. However, according to sources[4], Allegro had *fixed this vulnerability already in 2005*, which was revealed when checkpoint disclosed the 'misfortune cookie' issue, which was similar in nature.

---

[2] `https://www.theregister.co.uk/2016/12/08/talktalk_routers_may_be_botnet_imperva_says/`

[3] `https://www.allegrosoft.com/`

So what went wrong? Well, Allegro make money, like any other company, so they sell their software to IC manufacturers directly. In turn, these manufacturers develop a chip, and then sell/give away an SDK with the chips so that companies can quickly get products to market. As such, the problem was specifically that the manufacturer had never upgraded, so the product designers never upgraded, and these devices are in place for years and years - so a vuln from 2005 can and will be a major pain in the arse in 2017 because of this major problem in the supply chain **delay-line effect**.

You will find web servers you've never heard of (because they were discontinued in 2003, like D-Link's Boa webserver), DNS servers older than most of your underwear, kernel versions that belong in museums, and best not to mention what versions of busybox they are running.

Exposing to the WAN what shouldn't be on the internet

The above hack was caused by a UPnP payload working on a TR069 endpoint. *But how did this happen?* Well, the binary that ran the TR069 service also ran the local (unauthenticated) UPnP service on the LAN. As such, you could exploit the one through the other, as they were, in fact, the same.[5]

---

[4] `http://www.ispreview.co.uk/index.php/2014/12/`
`masses-broadband-routers-hit-misfortune-cookie-security-`
`html`

Other D-Link routers expose the ‘file sharing’ feature; any USB stick plugged into the router has its contents displayed for retrieval from the WLAN/LAN over a web service on port 8181. This service, however, is also displayed on the WAN device. It has a default login of user ‘admin’ and blank password.[6] It also has an auth bypass by forced browsing to `/folder_view.php` endpoint.

As is hopefully obvious, there were no need for the design and implementation 4AM decisions that lead to these issues.

Wi-Fi woes

Many attacks have been found against commercial Wi-Fi setups. But from this to this, it’s clear that all your Wi-Fi experience as a teenager has not gone to waste.

Remote Command Exec (RCE)

Many instances where you can send a command over the internet and run commands on the local device as r00t. This is quite obviously bad - and there are many sources for ‘what to do’ if you find this kind of vulnerabilty. Many devices also run everything (even web pages) through BASH scripts because then they don’t have the overhead of including a (probably as vulnerable)

---

[5] Echoes of the Hesperus vs. Phosphorus argument from antiquity rings loudly, here...

[6] To see this, a shodan search of ‘`WebServer port:8181`’ will display online devices.

PHP service. As such, injecting RCE strings all over is likely to find something, somewhere, on the web-app used to manage the devices.

Remote Code Exec

Likewise, connecting to some exposed service on the external device presentation and send crafted payloads to exploit BOF, format string vulns, etc. These service, if vulnerable, can easily be found using [shodan](https://shodan.com) and the like.

Cross Site Reqeust Forgery (CSRF)

Using CSRF to fire in a user's browser, connect to the ubiquitous web based apps/API's used to manage the devices, and either get RCE or directly manipulate settings is a common vulnerability I have found on firmware-based devices. The required CSRF-Token checks needed to mitigate this can produce lines and lines of code, which is a hefty price to pay on storage-light firmware chips (remember - the max for most firmware chips is 16Mb). This vuln is going to be a common theme.

Cross Site Scripting (XSS)

XSS is always bad, and if there is a way to compromise the device with, say, a stored or reflected XSS, the user's browser as well as the device can be compromised. I'm not going to write here on the evils of XSS as many others have done this before me! But your honed bug-bounty/web-CTF skillz are not lost here!

File Upload vulns

Linking into my nightmare CSRF scenarios, file parsing is almost always comically bad on devices, and I will put money on that if there's a file upload on a new-to-market device, it'll have some vulnerability related to either Local File Inclusion, Directory Traversal, or straight up pwning (as seen with a device that just unzipped a file as root into `/`, including the malicious `/bin/evil-ELF` I included).

XML/SOAP based attacks

More and more devices are using mobile based android/iOS apps to manage these devices. As cool as this sounds, these devices open up a SOAP/XML/JSON based API to talk to the app (rarely implementing any authentication), and UPnP attacks, enumeration, TR069 attacks and enumeration (and others) fall into this category, too - but see 'cloud' for more.

Authentication / Session management issues

Default credentials are everywhere - literally! `admin:admin` or similar über-weak credentials used to be a joke in the office, until it was everywhere on every device you could talk to that used some embedded platform. It is also a myth that most users even *can* change their passwords to these devices (many don't support this anymore - see 'cloud' below) so this is also worth reporting. Authentication information enumeration (usernames/emails) is also rampant, as well as authentication bypass - I remember doing a test on one device where just having a valid cookie (which you could guess or bruteforce to get) was enough to bypass username/password requirements.

Information disclosure

If the device passes through much of the user's data (as in CPE's) then this data is at risk if the device is - and so these devices can act as a source for further malicious activity (dumping CC data, seeing if someone is in the house for IPCams, etc.). Don't forget, setting malicious DNS servers is very, very easy on CPE's once you've compromised one, and most users (even tech savvy IT people) won't notice, not ever.

Additionally, if a devices offers up access to other services that they then manage (SIP/VOIP, FemtoCell or IPTV access) then disclosure of the internal mechanisms on these devices can reveal private keys for encryption, common shared secrets or credentials (often unencrypted/unhashed), certificates for authentication and encrypted access, DRM information and access methods, etc.

The 'Cloud'

...is just someone else's computer. But aside from all these issues (which are dealt with much better elsewhere) there is another concern. Amid this melange of written-at-4am-with-no-sleep and updated-when-West-Virgina-last-voted-Democrat software (see, US-relevant jokes... how international of me), introducing new and modern services is as bad an idea as you might thing.

All I can say is that there is little reason for a user to be able to change their firewall settings on their router from the other side of the planet (read; over the internet) but that hasn't stopped manufacturers' efforts to 'solve problems people didn't know they had', thereby creating more problems that nobody whatsoever needs.

5.2.1 Hardware Specific Issues

There are also some specific issues relating to hardware devices themselves:

More Supply chain woes

Example: U-Boot doesn't support secure booting. At all. Secure boot mode has been in ARM processors from 2005/6 time, but the most common bootloader in existence doesn't support it. This means that your bootloader probably doesn't support checking if the firmware is malicious. Again, tie this into a CSRF vuln (see above) and you can upload a malicious firmware file (I'll probably do a blog on how to make these).

But how is this a supply chain issue? Well, ARM is sold under license, and as such, many of the cheapest chips use licenses of older chip designs. So we see the precise same issue as above - Jazelle mode is an old ARM exec mode that supports native Java bytecode execution (*YES YOU READ THAT RIGHT*). It was 'replaced' by a new mode in ARMv7 called 'ThumbEE', but you'll still see Jazelle in millions of chips in the wild worldwide.

Development Headers

Kinda obvious from the outset, but the reason these are left on the boards is that the cost of tooling a ‘consumer safe’ board is very high. Prototyping can take weeks, and when you double that time to create a ‘secure’ version of a PCB layout, most project managers would shelve such an idea on the grounds of cost alone.

Also, ‘omitting’ a connecting resistor when going from dev board to production board *isn’t fooling anybody*...

Bootlog Oracles

Bootlogs are great. They’re on the list a little down from post-it notes and using your MAC address as the default Wi-Fi password[7]. The tell you processor versions, chip ID’s (they are often covered up with heat-sinks you need light munitions to get off), kernel versions, precise software versions, and tell you all the modes that things are running in/at/on. We will do some detailed commentaries on some bootlogs to show how this works later on.

5.3 Hardware Pentest Timeline - from Scoping to Reporting

This is a proposed ‘ideal’ timeline that you should work towards in order to do the most effective pentration test and security assessment of an embedded device that you can. This timeline is the product of professional experience and talking to other experts/testers in the field. Although you may not be able to execute all these steps exactly, it should serve as an overview for the ‘ideal’ hardware/embedded security assessment.

---

[7] `https://twitter.com/LargeCardinal/status/682591420969029632`

5.3.1 Scoping

Scoping Scoping Time

Have an alotted amount of time within which you can scope the device in question. It can take up to a week to fully analyse and research a board, and at least one example device must be provided for scoping. As such, this is the first step and only takes a few minutes to determine how long you will need to scope out the testing of a device.

PCB Analysis and research

Once you have this time, there three tasks, the first of which is PCB Analysis. The aim here is to gather information about the devices and IC’s in play, and how they work at a hardware level. Actions should include:

- Identifying all IC’s on a PCB
- finding and storing the datasheets for all identified IC’s (no matter how obscure!)
- making black-box diagrams for anything that isn’t obvious

- Identify developer headers and assess what they could be
- Using logic to solve identity problems

An example of the last step is as follows; An SoC had a digital signal out trace sent to a ‘black blob’ epoxy IC (these are the ones where the IC is smothered in black epoxy, and no clue of what the IC underneath does is available), and a line came straight out of here to an 8-pin SOIC speaker driver. It was therefore reasonable to assume, given this black blob spoke to the ‘signal in’ pin that drives the speaker, that this IC was a Digital-to-Analogue Converter (DAC). This made sense later when it was determined that the SoC had a built in ADC (Analogue-to-Digital converter) for recording sound, but no inverse audio-output channel.

Recording all this information - IC identitites, datasheets, diagrams, etc. will be very useful later down the line.

5.3.2 Testing

Testing should ideally have the following provided by the client:

- At least 2 devices for testing
- Firmware files
- Source Code for custom services (if code-assisted pentesting is in scope)
- Relevant technical documentation as available
- Disclosure of any known issues, should the client be willing

Ideally, the client should also determine any trophies - things that they are concerned about specifically. This is especially important if testing time is limited.

PCB Analysis and Research

It is highly likely that the device you eventually test is completely different from the one that you scoped. This is due to the fact that satisfying the requirement that a pentest needs to be provisioned is often put ahead, in time, of the device being finally ready - the idea is that you will test the device as soon as it completes its basic testing.

As such, it is possible that the entire PCB analysis may need to be redone once the test devices have been acquired. Allow time for this to be the case.

Dumping Firmware

Dumping the firmware, or analysing prepared firmware, is an essential step in assessing embedded systems.

Debug/Terminal access

This is going to be especially useful, and time should be allotted to specifically trying to access this functionality. If there are any in place restrictions, these should be documented, and their effectiveness should be determined - if you got past them, the manufacturer should know how it was done.

5.3.3 Reporting

Found Secrets/Trophies

Common secrets between different devices should be highlighted as a matter of priority, as they are probably in the wild. If it was possible to determine a unique secret from artefacts on the device (such as the MAC address of an interface giving rise to the default Wi-fi password), this should be made clear. Usual sensitivity should be exercised when dealing with these secrets.

‘Testing Narrative’ of Hardware Hacks

With regard to giving visibility, if required, the client should be able to get ahold of a rough testing narrative of all the things that were attempted against the hardware. This can be particularly useful for their own risk assessment exercises, so should be encouraged.

Chapter 6

Terminal Access through UART

Now that we've covered the motivation for *why* we should include low-level hardware hacking in our pentesting, and we've covered off the specific details elsewhere, it is now time to do a full run-through of what to do when you want to connect to a piece of hardware.

We'll start with the most ubiquitous and easy example - UART.

6.1 What we will need

Here's a list of the stuff you'll require to do everything in this chapter:

- Multimeter

- Soldering Iron (with solder and flux as required)
- Header Pins - usually set in plastic, 4mm apart
- Plastic Coated wire - esp. if you don't have the pins, cut to lengths with tinned ends
- Logic Analyser - connecting over USB, compatible with Saleae Logic
- USB to UART adapter - a CP2102-like adapter is what I use, or a bus-pirate
- A target board.
- Datasheets for the SoC/IC you want to connect to (may not be available)

6.2 Identifying UART

6.2.1 Identify and Connect to Development Headers

So, when they manufacture a PCB, they don't necessarily remove the development headers from the production models. What are these? Well, they are the parts of the PCB that would carry interfaces designed for use with test equipment the designers use in development of the firmware.

Remember, the PCB design comes in two stages - the first decides the layout of the parts, and correlates this to the layout requirements of a PCB that will fit inside the proposed product.

The second stage is where the firmware is written and developed so that the board works properly, and performs all required tasks.

Obviously, the first stage is costly - it costs a lot of money to prepare a PCB design, then have the prototypes made and populated, to then have access to a board you can develop on. Often, removing something like a debug header is not an option, owing a requirement to re-test the board. So to keep costs low, many prototype boards make it into production, minus the costly plastic headers that actually go on the board.

As such, this is what we want to find, so that we can do a little debugging of our own.

Identifying a Development Header

[Insert pictures]

Given we are looking for UART, we would expect there to be four pins, with two fine traces coming from two of them. As such, the ones with thick traces/connected to ground are likely VCC/GND, and the two with fine traces are signal lines, indicating they are likely Rx/Tx.

You can also look to the datasheets - usually available online - for the chip you want to connect to. Search the document for ‘serial’ or ‘debug’ or ‘uart’ and you’ll usually find some listing for the UART connection. Get the name of the pins, e.g. P_2_6 for Tx, say, and then search for the ‘pinout’ and look for that pin. There is usually a large dot, notched corners, or some other fiduciary mark on the chip to tell you which way round the diagram matches to the pins. You should then be able to follow the PCB traces and see if the development/debug access leads to a development head or not.

If you don't see what looks like a gap for a development head, then it is very possible that your SoC connects to UART through a 'bed of nails', and so you will need to examine the test pads. A bed of nails test rig is one where little springy brass collared pins called 'pogopins' are arranged in a fixed way to allow the developers to connect to a fixed set of test pads. As such, if your UART is sent there, then you need to solder onto a test pad, as described in the soldering skillz chapter.

Once you've identified the development header you want, solder a wire into each hole in the PCB, or onto each test pad, and get ready to analyse the signals with the logic analyser.

Analysing the Header Signals

We will need to use a multimeter to get a sense of which pin is what. For the sake of argument, we will assume that we have four pins, which we will write $\begin{array}{|c|c|c|c|}\hline a & b & c & d \\ \hline\end{array}$. Find some ground point (either labelled, or use the one connected to the main ground off the power supply unit (PSU).

1. Use the multimeter continuity setting (or the diode setting) to check which of our candidate pins is GND. You should see full conductivity to the ground point, as if there is nothing in the way (should be a close reading to touching the probes together). Label this on some diagram, for example if it's the last pin, label as $\begin{array}{|c|c|c|c|}\hline a & b & c & - \\ \hline\end{array}$)
2. Next identify the VCC line - this should be the pin with the highest voltage whilst the device is running. Chips run at

3.3V, but generally you'll see 3.2V or thereabouts. As such, two have a lower voltage than the third, that is a stable 3.3V from the PSU assembly. Label this on your diagram, e.g. if it is the port far left, we update it to $\begin{array}{|c|c|c|c|}\hline + & b & c & - \\ \hline\end{array}$.

3. We can now make the safe assumption (which we will test) that the remaining two pins are Tx and Rx. To work out which is which, you may have noticed some variance on the voltage of one of the pins when finding VCC. If you did, then that's probably a Tx line.[1]

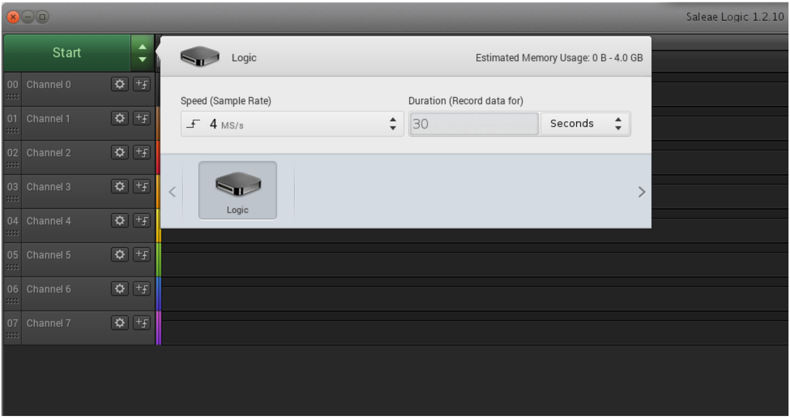
We are now in a position to use the logic analyser with the Logic software to The general methodology for analysing the signals from a device is as follows.

Connect up the analyser - Connect GND to the relevant pin on the analyser. You can connect VCC to a channel if you aren't certain it's the power line. This will now let you use the logic analyser to work out what is what.

Connect the pins to analyse - We have two, so we'll use channel 1 and 2 in our example. Now configure the sampler as in Figure 6.1. This will work for most UART implementations in the wild.

---

[1] Multimeters use Root-Mean Square (RMS) chips to 'even out' the peaks and troughs of a signal to give a true level reading. As such, it would show lower voltages, such as 2.7V or briefly 0V, as it tries to interpret an digital signal.


Saleae Logic 1.2.10
Start
00 Channel 0
01 Channel 1
02 Channel 2
03 Channel 3
04 Channel 4
05 Channel 5
06 Channel 6
07 Channel 7
Logic
Logic

Estimated Memory Usage: 0 B - 4.0 GB

Speed (Sample Rate): 4 MS/s

Duration (Record data for): 30 Seconds

Figure 6.1: Setting the Sample Rate in Saleae Logic

Once this is setup, start the capture, and then immediately start the target device. Once it is finished you will see a caputured digital signal, as in 6.2. This will then need to be run through the analyser to get out the baud rate.

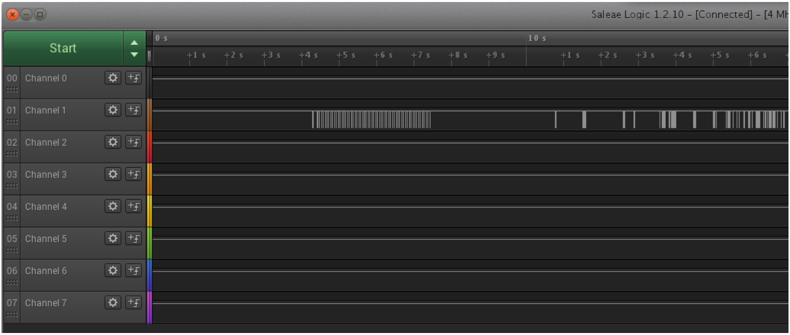
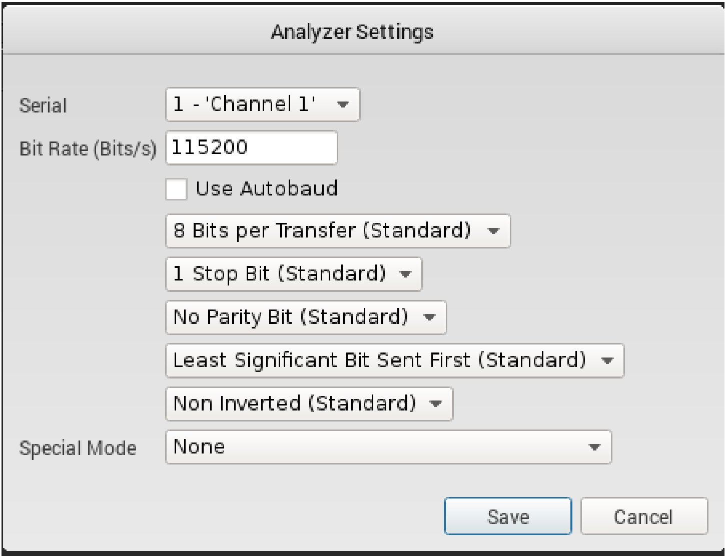

Saleae Logic 1.2.10 - [Connected] - [4 MH
Start
0 s
10 s
+1 s
+2 s
+3 s
+4 s
+5 s
+6 s
+7 s
+8 s
+9 s
+1 s
+2 s
+3 s
+4 s
+5 s
+6 s
00 Channel 0
01 Channel 1
02 Channel 2
03 Channel 3
04 Channel 4
05 Channel 5
06 Channel 6
07 Channel 7

Figure 6.2: Captured Digital Signal

ProTip: If you get no signal from either, or signal from one but you can't connect, double check the traces. They may have omitted a resistor from the production model to stop people connecting to the debug interface (or just to save costs).
Double check the traces leading to/from the SoC to your, and look for gaps. If you find one, use a dab of solder to carefully short the gap, allowing you to connect.

Run the Analyzer Now we are ready to analyse the signal we have. On the right of the main screen in Logic you will see a small box for 'analyzers'. Click the cog icon, and you will see the analyser setting screen, as per Figure 6.3. This will then allow you to load an analysis of the signal for different baud rates and settings. Nominally, all UART systems in the wild that you will encounter will tend to use `8N1` as the data settings (see the UART protocol description for more details about that). As such, our only task is to work out what the baud rate is.



Serial: 1 - 'Channel 1'
Bit Rate (Bits/s): 115200
☐ Use Autobaud
8 Bits per Transfer (Standard)
1 Stop Bit (Standard)
No Parity Bit (Standard)
Least Significant Bit Sent First (Standard)
Non Inverted (Standard)
Special Mode: None

Figure 6.3: Analyser Settings - note that the baud rates are free-form text.

The most common baud rates, as mentioned in a previous chapter, are as follows:

- 115200
- 9600
- 57600
- 19200

- 38400

Once the successful one is found, you should see some meaningful output mapped above the signal when you zoom in on Logic. A successful analysis with the correct baud rate will yield an output that looks sensible, as found in Figure 6.4

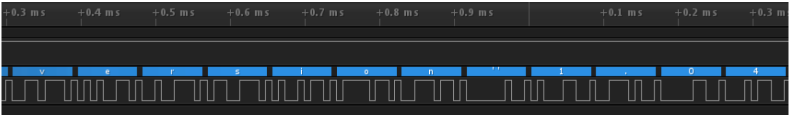


Figure 6.4: Successfully analyzed signal showing `version 1.04` being read from the data.

We have now successfully found our pinout for the development header, as well as our connection settings.

6.3 Connecting to UART

NB - For the sake of this section, we will assume that the pinout was $\begin{array}{|c|c|c|c|}\hline + & Tx & Rx & - \\ \hline\end{array}$, and that the baud rate was 115200 (the most common).

6.3.1 Testing Tx

We are now ready to connect for the first time to our device. We now want to wire up our USB to UART adaptor - for ease, I will assume it's a CP2102 (the most common IC used on these boards) and will refer to it by the chip name. **NB** - ensure that the device is off!

Connect GND to the GND pin on the CP2102, and take the Tx from the target pinout, and connect that to the Rx on the CP2102. Now run the following command (as root):

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

Here, `/dev/ttyUSB0` is the CP2102 device[2] and 115200 is the baud rate we identified in the previous section.

Now power on the device and you should see text output in the screen terminal. It should look like a boot sequence log - that's because it is! If you're seeing this, then congratulations, you've successfully connected over UART!

6.3.2 Connecting fully with Tx/Rx

We are now ready to try and get a shell. Power off the target device, and disconnect the CP2102. Connect the remaining Rx pin on the pinout to the Tx on the CP2102. Reconnect the CP2102, start screen (the command will be the same) and power on the device. If you see some generic `Please press a key to interrupt boot process...` try it. You should either get dumped into a bootloader shell, and can access memory, or if you allow it to continue (power cycle the device if needed) you may get dumped into a root shell.

This is the debug shell that the developers used when building the firmware, so now you're ready to start testing.

---

[2] If you're not sure, you can check by reading the `dmesg` log after connecting the CP2102.

6.4 UART Situational Awareness

There is a wealth of information you can glean from the boot sequence dump (called a 'bootlog'). Items worthy of note include:

- Bootloader name and version (Common ones are U-Boot and HiZ)
- CPU/MCU architecture (ARM/MIPS usually)
- Linux Kernel version
- busybox version (discussed below)
- Other software names and versions - check against known CVE's
- Look for locations of important scripts (like `/etc/rc.d/rc0.sh` or the like)
- Device mappings should also be checked (like whether the UART console is mapped to `/dev/ttyS0`, `/dev/ttyS1`, `/dev/console`, etc.)

If you are a little lucky (not all devices do this, but most do) then at the end of this you will see a root shell. You may have to login, but that login is usually the same as the default for the management web application, as a rule of thumb. We will discuss how to use this shell and what to do with it once we have given a methodology for dumping flash ROM data using SPI over a bus pirate.

Chapter 7

Fimware Dumping through SPI

SPI (Serial Programming Interface) is as ubiquitous as UART when it comes to prevelance in the IoT/embedded hardware world. When used as a communications protocol, it's primary use has been to interface and SoC with a Flash ROM IC. Although it can be used in other areas - many LCD screen breakout boards use SPI as their communications protocol with Raspberry Pis, for example - this is the most common form you will find SPI in use on production PCB's.

7.1 What we will need

Here is a list of the stuff you will need to do everything in this chapter:

- Multimeter
- SOIC Clip or IC Leg Taps
- Bus Pirate - or other SPI compatible protocol board[1]
- `flashrom` installed - or the software that works with another chosen SPI protocol board
- `binwalk` installed
- Optional items:
    - Logic Analyser with Saleae Logic software

7.2 Identifying SPI

7.2.1 Standard Analysis

Analysis of the PCB should identify one chip that is an 8- or 16-pin IC that has a part number corresponding to a datasheet that tells you this chip is a Flash ROM Storage device.

It must be noted that not all 8-pin SOIC IC's are ROM chips. Many power convertors, speaker driver/op amp IC's, small MCU's (Microcontrollers) all come in these packages. You **must** confirm that the chip is indeed a ROM chip and supports SPI before continuing to connect it to a Bus Pirate. If you don't, bad things will happen.

---

[1] A 'protocol board' is just a board that speaks to your computer and lets you communicate with a target over the defined protocol.

ProTip: Flash ROM SOIC's tend to be a little 'thicker' and more square than the other types of chip using an SOIC package. This is due to the silicon die needing a large square NAND flash storage area. As such, the IC's themselves tend to look a little different from the other 8-pin SOIC's in particular.

Once you have identified the pinout from the datasheet, you're ready to connect to the device and dump the contents!

7.2.2 What to do when Standard Analysis fails

First, with the device powered on, identify the VCC pin with a multimeter - this can be done by putting the negative (black) probe on GND (say the collar of the power supply connector that goes to ground) and trying each of the pins in turn. As a rule of thumb, it is generally the pin opposite the pin next to the little circle (the fiducial marker) - if the circle is 'top left' then this pin is 'top right'.

Now disconnect the unit and use a multimeter in continuity mode (or diode test mode) to identify the GND pin. This has to be continuous from all GND terminals to main ground, so using a known GND point (like we did earlier) is best. As a rule of thumb, this is generally the pin at the opposite end of the chip, on the side with the circle (fiducial marker). If the marker is 'top left' the this pin is 'bottom left'.

You now have a decent indicator for the layout of the IC. This should help you work out if the pinout for this chip is similar to others - it is best to see if you can get the datasheets for chips in the same series, if not the exact one. Searching online for these datasheets may be as fruitful. Atmel ROM IC's, by and large, do not follow the diagram below.

Following from my 'rule of thumb', here is a standard pinout that works on most flash ROM IC's I have encountered personally:

Bus Pirate	Flash chip				Bus Pira
CS	1	CS	VCC	8	3.3V
MISO	2	DO (data out)	HOLD	7	3.3V**
3.3V**	3	WP (write protect)	CLK	6	CLK
GND	4	GND	DI (Data In)	5	MOSI

Figure 7.1: Generic ROM IC Pinout - the numbers are just IC pin numbers **See the IC datasheet for whether these should be high or low in order to read the contents.

7.3 Dumping firmware with SPI

Once we have our proposed pinout, we are ready to wire up the chip!

Ensure that the device is powered off! Else, you will have the SoC to compete with and all manner of mayhem will be set loose.

Following the wiring that you have identified, as the SPI pinout descriptions line up with the markings on the bus pirate.

- MOSI goes to MOSI
- MISO goes to MISO
- SS/CS goes to CS (chip select) on the bus pirate

- CLK goes to CLK.
- Now wire up VCC for power and GND.

ProTip: Flash ROM's can operate at 3.3V or 5V. Check the datasheet or use a multimeter to determine which is the correct voltage, else you either won't get a signal, or worse, could damage the chip by over-volting.

Now we can attempt to dump our firmware. Using a bash prompt, run the following as root:

```
>$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,
    spispeed=1M -o ./firmware-dump.bin
```

NB - if at first it doesn't work, don't worry. Try it again. If it still doesn't work, adjust the `spispeed` parameter. The supported speeds for bus pirate in the flashrom source are:

- 30k
- 125k
- 250k
- 1M
- 2M
- 2.6M
- 4M
- 8M

So, if you are having problems with a particular ROM chip, start by lowering the `spispeed`. Upping it to 2M is supported by most ROMs (and may be a hard limit on the bus pirate itself), and will dump the storage faster.

If it is still not possible to dump the ROM file out, then there are a few possibilities:

1. You may have made an error in your wiring
2. You may have followed the wrong datasheet
3. You may have a ROM chip that has specific requirements - you may need to adjust the buspirate firmware settings
4. Worst case, you may have killed the chip by accident during handling - use a logic analyser when the device is running (if it runs)

If all has gone well, you now have a file `firmware-dump.bin` that you are ready to analyse.

7.3.1 Firmware Dumping through UART and `tftp`

If all else fails, then it is sometimes possible to dump firmware through the UART connection. This is documented here only for the `U-Boot` bootloader, although similar things are possible on others.

The basic methodology is as follows:

1. Start the interface for UART (screen or minicom) in logging mode - this is important.
2. Load the bootloader into debug mode (usually by pressing any key during boot).
3. Determine (through the `help` menus) whether there is a way to display memory.
4. Ensure that the firmware image is loaded in memory.

5. Use the memory display functionality to dump the contents of memory to screen.
6. As we're logging, this will now be logged to your local drive.

Worked Example

Here's a walkthrough for U-Boot, using screen, with the decoding at the end.

1. As root, run `screen -L screenlog.log /dev/ttyUSB0 115200`
2. press any key to interrupt booting
3. either using the bootlog or `sf probe`, get the addresses and offsets for your flash image
4. Load flash partitions into memory using `fload`, or if this fails, try `sf`
    - for example `sf probe 0;sf read 0x82000000 0x40000 0x370000`
5. Dump memory to screen using `md` (memory display)
    - e.g. `md 0x82000000 0x40000`
6. Copy the escaped hex into a separate file.
7. Hex that is in a hexdump format can be decoded into a binary file using `xxd -r`
    - e.g. `xxd -r text-dump.txt > firmware-dump.bin`

Further options

If this fails, see if your u-boot supports tftp, and follow these steps:

1. As above, use `sf` and/or `fload` to ensure that the ROM images are in memory.
    - To be clear, I mean; `sf probe 0;sf read 0x82000000 0x40000 0x370000`
2. using the `tftp` command, load the firmware image you want over the wire
    - Start a local tftp server
    - start a local DHCP server if you can, else, set static IP's that are compatible with the subnet used by the device
    - Attempt to load the file over tftp; `tftp 0x82000000 romfs.cramfs 0x370000`

Chapter 8

Firmware Analysis

We will assume that the dumped firmware file is called `firmware-dump.bin`. This can either be an in-memory dump off a device, but can also be from a firmware file provided by the client or one that you found online - generally the ones online are just replacements for the ones that are held locally; the binary file gets written as-is to the ROM chip.

8.1 Extracting Firmware

First, run binwalk on the file to see what the likely interesting offsets are. This can be done via

```
binwalk -e firmware-dump.bin
```

This will create a subdirectory `./_firmware-dump.bin` in which you can find any files that binwalk recognized and dumped or the contents of compressed directories and filesystems that binwalk was able to recognise and extract.

If this produces nothing, then run `binwalk ./firmware-dump.bin` to get an overview of the technical details and offsets. You can then use `dd` like so:

```
dd if=./firmware-dump.bin skip=<offset address>
    count=<number of bytes to dump>
```

Remember, the `count` parameter takes decimal input, not hex, so make sure to use these. Thankfully, `binwalk` outputs this as well, which is handy.

ProTip: If you are not sure what to make of a file and binwalk is not making sense of it, look at the bootlog. This will usually contain the filesystem format, start addresses and block sizes, as well as compression schemes involved.

8.2 Trophies and Loot

Here are the things you should keep an eye out when parsing through extracted files from a firmware image. These artefacts are particularly useful when testing, as they may give you some clue as to how to overcome a particular login/parameter, and are usually universal for all devices of the same model number or series.

General tips when hunting for such treasure:

- Keep a note of *what* was found, so you can check if it changed later
- Keep a note of *where* it was found for use in later testing

Configuration Files

Many firmwares, espeicially for `RomPager` based firmware files, there is a file `/etc/romfile.cfg`, that can also be found at `/var/romfile.cfg` in some instances. This file contains all the details required to setup the system on the fly. When the system is booted this file is parsed out, and its contents are used to populate various BASH script `.sh` files, as well as populate configuration files for wpa_supplicant settings, DHCP ranges, etc.

As such, this is a good example for understanding the kind of thinking that takes place when firmware files are put together. There are near-infinite ways of achieving the same thing with the same setup, so it would be remiss to say “This is how it’s done!”, when it clearly may not be the case. Instead, here are some tips for

Passwords/Secrets

The following command, run in the directory where you store the extracted files, will find all sorts:

```
grep -RiE --color 'pass|usern|config|private|key'
```

Pulling `/etc/passwd` is a good idea - cracking the passwords[1] therein may reveal default user accounts that can access SSH/Telnet/management ports/managment web application/etc. It also may give you a clue as to how authentication is done on the web applications used to manage the app - very often, the

---

[1] Generally passwords are hashed with one of the earlier `crypt` implementations - usually `$3$` or `$3$`. No joke.

`/etc/passwd` file, or some other localised `passwd` file, is used for all authentication sytemwide, or for multiple subsystems.

Likewise, parsing for common developer variable names such as `secret_key` or `api_key` may reveal uniform secrets that are used across all these devices.

The same can be done for DECT phone secrets on CPE's that can talk to wireless phones (which would enable call snooping), and for media access/Digital copyright keys on devices that access media, like IPTV's. Also, keep track of what private keys for certificates are kept, and what they look like they are used for. For example, some systems require authenticated access to some private subnet for hosted services, with access being restricted to any devce that has a particular certificate - if you now have said certificate, you have open access to these 'restricted' systems.

NB - Recording the locations for these will be useful in later testing, too, so keep note!

BASH Scripts

When an embedded device is loaded, chances are everything is managed/updated/configured by BASH scripts. As such, knowing where scripts are, and where they get their parameters from is really important. This is how many RCE's are found in embedded systems.

Binary files

Knowing the location of important binaries is very useful in later testing. Keep track of what binary files are where. Also, if a binary file does something specific but is not named properly, note stuff like that down. You *will* forget, and *will* get frustrated trying to remember later!

We will cover the disassembly of ARM/MIPS/XTensa/etc. binaries later on.[2]

8.3 Correlating with UART

Having access to a UART interface to some SoC will greatly increase the chances of you making full sense of the way a system works, and increase thereby the likelihood you will make significant advances in determining the vulnerabilities on the attack surface of the device.

When checking the actual locations of files, it can be handy to use the UART interface to make sure that files are indeed where they are in the initial firmware. Often, files are copied and then manipulated - in our example above, the `/var/romfile.cfg` file is soft symlinked from `/etc/romfile.cfg`. This can take hours to decipher looking at the firmware dump itself, but is abundantly clear when you have access to the UART terminal.

---

[2] Fun Fact: if you see lots of hex opcodes in a file that begin `0xDx` or `0xEx` that are close to each other (or line up in a hex editor into columns), then chances are it's ARM in thumb-mode, as these are the opcodes for the branching instructions.

The UART will also tell you what devices are where - mappings such as `/dev/ttyS0` is a common mapping for the serial port. Using `echo` to send some data to that location should reflect back to you. Once confirmed this can be used when finding RCE vulnerabilities (covered in the next chapter).

8.4 Reverse Engineering Firmware and Binaries

Firmware generally splits into three categories; those running just straight firwmware (single purpose code, without any OS or filesystem to speak of), those running as some sort of RTOS (e.g. FreeRTOS), or those running some known kernel and filesystem (sometimes running in an RTOS-y manner - as found with Linux on many home routers/CPEs).

In this section we'll cover basic reverse engineering of both - in IDA and Radare2/Cutter where appropriate.

8.4.1 Reverse Engineering ARM/MIPS Binaries

These are actually quite straightforward to reverse engineer. Both ARM and MIPS instructions sets are well supported by both IDA, R2, and other disassemblers. As such, we will leave this to the chapter (to be included) which will deal with binary reverse engineering for embedded systems more specifically.

8.4.2 Reverse Engineering a Firmware 'Blob'

Sometimes, you just get a load of firmware - no filesystem, just code that runs and that is it. This is by far the most common thing to find in IoT/embedded devices. But how to reverse engineer it? Here are some pointers for the curious:

- Find the datasheet for the microcontroller you want to target and get the memory address values - these tell you where, for example, the first UART port or the third I^2C controllers are placed in memory for the device.[3]

- Once you have attained this information, seek out functions and name them appropriately - here are a few possible examples to get you thinking along the right lines:

    - If a function reads from a UART Rx buffer and puts this data into another memory location, then this is a `UART_Read()` kind of function.
    - If a function writes a byte from a register to a UART Tx buffer, then this is a `UART_Send()` type of function.
    - If a function reads many blocks from an SPI bus, then this is possibly a `firmware_copy()` kind of function
    - If this is called at the start and then has multiple calls to the ARM `CRC32W` opcode, then this is likely where a firmware check is being done...

---

[3] For an example IDA memory mapping config file for AVR microcontrollers such as ATMega328, see: `https://gist.github.com/extremecoders-re/8d3e9b846a6ec883e5ae3b2bccf5cc88`

A very nice bug that was found in the NXP LPC1348 bootloader (and will be discussed later for how it can be used to bypass CRP) by Chris Gerlinsky[4] - here, it would only retrieve the security bits one time, and had fixed values for firmware protection modes. This lead to his using this to enable a glitch at the moment this read is undertaken - thereby disabling CRP protections. But more on this later...

Once you have this mapping done, and have worked out a few of the main functions, you are ready to commence reverse engineering in the 'normal way' - as you would if you had a file with an ELF header that you were more familiar with. Again, these specifics will be dealt with more fully elsewhere.

---

[4] https://twitter.com/akacastor

Chapter 9

Radio Hacking

9.1 Wireless Communications - the Backbone of IoT

One of the things that makes IoT devices (and indeed, any embedded system) so useful is the capability of acquiring some sort of data and then relaying that to some central assessment destination for analysis, or taking some sort of command and then actuating that into a real action across multiple devices in multiple locations/spaces.

Although we can achieve much with wires, the advent of wireless communications has facilitated a multitude of easy-to-include, ready-made wireless solutions for sharing data between devices. As such, any consideration of IoT security must necessarily include a treatment of radio hacking, from basic WiFi vulnerabilities we all know and love, through to BLE ubiquity and hacking, and rounding off on the more IoT/embedded specific protocols; LoRaWAN, SigFox, ZigBee, etc.

9.1.1 Radio Basics

What is a ‘Radio’

Wired communication protocols (such as the serial ones like UART, SPI, etc. we saw earlier) have their range determined by properties of the wire, the power used to send the signal down said wire, and external parameters such as noise, interferance, temperature, etc. Taking all these together defines how well your ethernet will work, whether you can run that serial cable from the basement to the top floor terminals (back in the day...), how fast you can communicate down that wire, etc.

Similarly, wireless protocols are governed by similar parameters, except we don’t have actual wires. Instead, we have a suite of frequencies - from ˜10MHz through to 5GHz and beyond. As society, we have split this up into bands, and some of these are free to use for anyone - such as 433MHz, 868MHz, 2.4GHz bands - and some are very strictly controlled - such as 800, 1800, and 2600MHz bands for use by 4G/LTE communications, for example.

The propagation of a wireless signal is governed by a few easily identifiable factors - these are factored into equations for what is called ‘Path Loss’:

- Antenna length/height
- ‘line of sight’ between transmitter and receiver - how clear is the path between Tx and Rx

- The distance between a transmitter and some receiver

Signal strength is measured in dBi, or Decibel Isotropic - and yes, this is a cousing of the decibel system used for measuring sound signals in audio electronics, usually dBu or dBv in that case. dBi is the gain measurement of an antenna compared the the 'isotropic antenna' - although this sounds like something you got threatened with in pre-calculus, the isotropic antenna is the 'idealised most perfect antenna' - given such a beast is physically impossible, we measure antennas on how close they come to imitating the gain of this perfect antenna.

Worth mentioning common terms you've seen before, e.g. in UART, come from radio - such as using TX for 'transmit' and RX for 'receive'.

Software Defined/Configurable Radios

In order to access this seemingly magical world of EM waves traversing the air, you're going to need a few tools. These tools take the form of Software Defined Radios (SDRs), or sometimes the related (but distinct) Software Configurable Radios (SCRs). In short, there are a few classes of tools available, ordered from cheapest to most expensive:

- **VGA to USB SDR** - costing as little as €5, these little VGA chips don't have any cache, meaning that the chip can be pulsed raw from the USB socket by sending carefully crafted packets to the IC. This rudimentary transmit-only radio can achieve TX of multiple frequencies, though its reliability is clearly better at the lower end - sub 1GHz.

- **RTL-SDR radio** - usually packaged as ‘TV Tuner SDRs’, these RX-only devices are very cheap, usually around €10, and have a reasonable range and reliability. Can be used to read from around 100MHz through to 2GHz reasonably reliably (meaning they can be used as passive IMSI catchers, or passive 1090 transciever aviation radar, for example).

- **Sub-1GHz SCR** - based around the TI CC1111 chipset (found in the [in]famous IM-Me devices, and many others) the Yard Stick ONE comes in at around €110 and are very effective. Being an SCR means you cannot configure it to use any modulation, but rather the ones it has pre-programmed - but it has enough for serious usage in testing the radios of common household devices.

- **Desktop USB SDRs** - A ‘proper’ SDR can be picked up for as little as €80 in sales, up to a few hundred Euros for a HackRF ONE or BladeRF device, all the way up to €2000+ for an Ettus Research SDR. These classes of SDR have full configurability, and ranges from low MHz all the way up to 5+GHz - serious amounts of usage for serious amounts of money.

It is worth noting that you should acquire the SDR that is most appropriate for what you want to do. A good example of this is the HackRF vs. BladeRF - on the face of it, the HackRF has a better frequency range and features, but it is half-duplex - not so useful if you want to MITM a protocol such as GSM/LTE - to do so, you would need 2 HackRFs, which is a bit pricey. However, the BladeRF, though not as wide a frequency range, is full duplex

- it can simultaneously transmit and receive. As such, It is always best to choose the right SDR tooling for the job at hand, so survey the market and assess your exact requirements early on in the assessment preparation phase.

ProTip: When it comes to any non-ISM[a] band, there are legal restrictions. For 3G/4G/LTE, these frequency bands are heavily controlled in every country on the planet. Interfering in these bands is particularly illegal - mainly due to the fact that you incapacitate people from making emergency calls vey easily. In the UK, this is governed by the Wireless Telegraphy Act 2006, for referene. **Proceed with caution.**

---

[a] The ISM bands are the ones that we mentioned earlier - where anyone can transmit or receive in without a license.

9.1.2 Radio Terminology

General Terms

Just so we're clear, here are some common terms used in radio communications, and some brief explanations about what they mean:

- **Frequency** - literally, the number of cycles (specifically, the full up-and-down of a waveform) per second, measured in Hz - 1Hz = 1 cycle (one up-and-down) per second.

- **Modulation** - the way in which we communicate data over a given frequency/frequencies

- **Channels** - For any given frequency *band*[1] there are usually defined several frequency ‘channels’ which are distinct, if sometimes overlapping, sub-bands identified to prevent interference between devices
  - The ISM 2.4GHz band - from 2400MHz through to 2490MHz - can be divided up into bands in many ways.
  - WiFi divides the 2.4GHz band into 14 channels; channels overlap with each other, but are designed that channels far apart will never interfere - as such, channels 1, 6, and 11 are most commonly used as they have no overlap with each other, so devices can talk freely within the ISM band.
  - Bluetooth and BLE split the 2.4GHz ISM band into 40 distinct channels, and uses lower-capacity modulations to communicate over such narrow bands. They also channel hop - Bluetooth negotiates this, but BLE has it predefined - between the channels in order to minimise interference (both from and to the bluetooth/BLE communications).

Modulation Methods - an Overview

There are broadly 4 types of ‘keying’ - ways of represenging ‘1’ and ‘0’ in radio communications. We’ll summarise them here: [TODO - add diagrams for each...]

On-Off Keying (OOK)

---

[1] Not as in rock bands - more as in ribbon bands...

This is probably the simplest - you represent a ‘0’ with silence, and a pulse at a given frequency indicates a ‘1’. Clearly, this has to be clocked/sampled independently of the signal, but it is the simplest digital radio transmission to enact - it’s the digital version of analogue morse-code.

Very simple, very low power, but also, not incredibly reliable, and prone to being messed up by reflections (off a bulding or other generic Large Thing near the receiver). This is the kind of modulation used by garage doors, doorbells, etc.

Amplitude Shift Keying (ASK)

Amplitude Shift Keying, or ASK, is the digital version of Amplitude Modulation in analogue transmission - yes, the same AM that you get transmitting on radios (normally received on a ‘crystal’-based (read: diode based) radio set ‘back in the day’...). Technically, OOK (above) is a slightly extreme version of ASK - where the gain is pulled down to zero to represent ‘0’.

As the name would suggest, the amplitude shifts up for a ‘1’ and down for a ‘0’ - this means, quite literally, that the 1’s are louder than the 0’s, and that’s how you can decode the signal. These are coded by adding gain amplifiers to the carrier wave when a 1 is present on the data bus.

Frequency Shift Keying (FSK)

This is the digital version of Frequency Modulation in analogue radio transmission - and yes, this is the FM that you listen to radio stations over, normally.

This time, there are two carrier frequencies - the lower frequency usually representing ‘0’, and the higher frequency representing ‘1’. These are effectively switched by the radio given an input of 1 or 0, and as such, it is very efficient and easy to implement as well - however care has to be made to ensure the phase of the two oscillations is maintained to prevent ‘broken’/discontinuous waveforms.

There are 2 variants you will find - 2FSK and 4FSK. The one I just described is 2FSK. 4FSK is the same, but with 4 frequencies, one each for ‘00’, ‘01’, ‘10’, and ‘11’. There are also several ways of generating the waveform - one is called Gaussian FSK or just GFSK, where the waveform is passed through a Gaussian filter to ‘clean it up a bit’ and smooth it out before transmission. GFSK is the primary modulation used in DECT phones, Bluetooth, BLE, LPRF, Z-Wave, and more.

Phase Shift Keying

Phase shift keying, or PSK, may sound like how door locks work on a sci-fi teenager’s bedroom, but in actuality it’s another kind of data keying for digital transmission.

First, a note about phase - this is the measurable difference between the peaks of two waveforms of the same frequency. If you have two waveforms 180° out of phase, then you get no sound at all from mixing them - waveforms that are only slightly out of phase give all sorts of weird and wonderful tonalities and timbres to instruments; this is a problem (or a feature) when recording electric guitars sometimes.[2]

So, with BPSK - Binary PSK - we have that two phases of waveform represent ‘1’ and ‘0’, and these waveforms are shifted around based on the data input values. As such, data can be transmitted at one frequency and at one amplitude, yet data can still be recovered from the (discontinuous) waveform.

---

[2] That said, the iconic sound of the Greeny/Moore Gibson Les Paul was entirely due to this property - they’d put the pickups on the guitar out of phase to get the sound effect they popularised.

The big-brother of BPSK is QPSK - Quadrature PSK. This has a 4-node complex-plane constallation mapping that defines data transmission, not too dissimmilar from the 4FSK above.[3]

PSK-based modulations are very reliable, but harder to implement in hardware. As such, they tend to be used in applications where higher data rates are required. That said, it is a strict 'subset' modulation to QAM.

Quadrature Amplitude Modulation

QAM is the hybrid between QPSK and ASK. But how does this work? Well, first we need to do understand a little about complex numbers.

Complex numbers are just numbers of the form $a + ib$ where $i = \sqrt{-1}$ - this last bit is the 'imaginary' numbers bit, but really it is just a number space that doesn't 'make sense' in the normal number line (as no number times itself gives -1) - but the solutions to equations often find themselves branching out into complex number spaces. As such, you can make an x, y graph, just like in school, only with the 'real' numbers going left-to-right, and the 'imaginary' numbers going up-and-down.

To spare you a *lot* of maths, I'm just going to tell you that 'spinning stuff' on a complex plane is the same thing as shifting waveforms forwards and backwards - that is, rotating around the unit circle on the complex plane is akin to manipulating the phase of a wave.

Last thing to realise is that the amplitude of our wave corresponds to a 'bigger/wider' circle on the complex plane.

---

[3] If that sentence made no sense, then don't worry - either google more, or forget you read it :P

That’s it - maths over. That’s all you need to know.[4]

Now that you know this, you can see how the phase of a wave (rotation around the central origin) and the amplitude of a wave can be coded to retrieve bits from the diagram in figure 9.1 - called a ‘constellation diagram’.

NB - For reference, the QPSK we discussed earlier would be equivalent to the 4 inner nodes in this graph (labelled 0101, 1101, 0111, 1111) alone - without the surrounding extra 12 nodes.

---

[4] Trust me, I’m a mathematician...

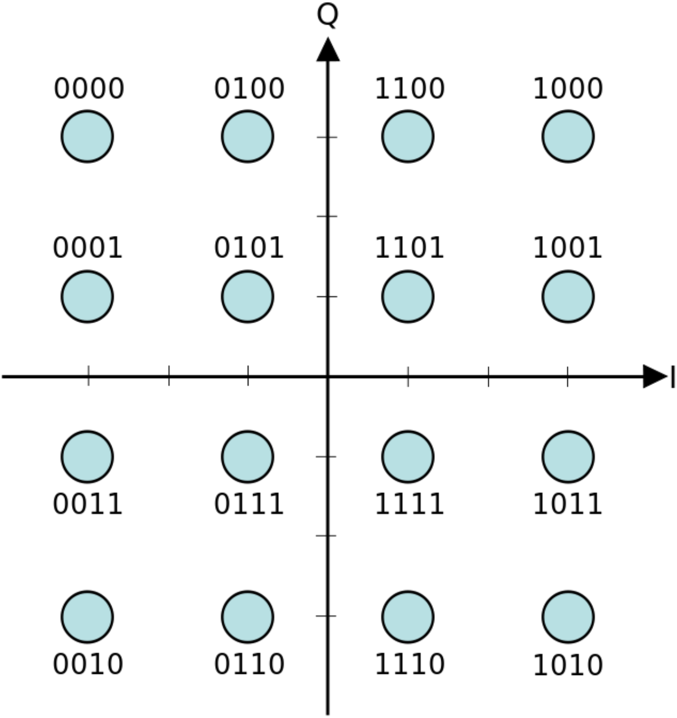

Q
I
0000
0100
1100
1000
0001
0101
1101
1001
0011
0111
1111
1011
0010
0110
1110
1010

Figure 9.1: QAM16 diagram - c.o. Wikimedia; `https://commons.wikimedia.org/wiki/File:16QAM_Gray_Coded.svg`

As such, we can combine the ‘rotational’ aspects of QPSK with measurements of gain level from ASK in order to have a setup corresponding to every node on this diagram. Thus, each snippet of our radio transmission coresponds to 4-bits in this new setup - hence the name, QAM16.

Fun Fact: QAM, in its various guises, is the base modulation used by WiFi, 3G, 4G/LTE, etc.

Additionally, you can have a 4×4 grid in every quadrant, giving you what is known as QAM64 - this is used in LTE, in particular.

9.2 The Radio Attack Surface

Now that we have given a solid coverage of (quite a bit of) radio theory, we should get down to business - *How can we use this knowledge to hack radios used in IoT?*

9.2.1 An overview of attacks

With IoT/embedded systems, you will find that you have to get closer for some radio implementations (such as BLE) or can attack the presented radio interface from some distance away (as with SigFox or LoRa). Here will will outline some of the attacks that are possible against radios of any type:

- **Replay Attacks** - from opening garage doors to opening cars to deactivating alarm systems, radio replay attacks are often found on systems that have a simple switch logic that reads a specific code in order to switch state - that is, the same radio pattern will lock as unlock a door, say.
    - It is worth emphasizing that, even with encrypted communications, unless your key changes or you use

a stream cipher, packets will always look the same. An attacker might not know that some packet indicates a temperature of $15°C$, but they can deduce it means ‘less than $20°C$.

- **No Encryption** - this peculiar folly is where the designers, assuming nobody would think to use an SDR (or not being aware how easy they are to acquire), neglect to make use of any encryption during transmission, rendering the information readable to anyone who cares to listen in. This is very common on older systems, such as early pager networks, for example.

- **MITM Attacks** - as with other hacking, radio hacking has ways of man-in-the-middle’ing radio traffic in order to defeat rolling codes[5], usually for the express purpose of fooling the receiver into thinking a command is legitimate by capturing one code, whilst jamming the receiver - for more, see Samy Kamkar’s ‘RollJam’ project.[6]

- **Information Disclosure** - some radio protocols are very helpful, and sometimes data can be disclosed or access can be granted unwittingly. This is common with naive BLE stacks, for instance.

---

[5] Codes that are sequential, usually based on a shared key known only to the receiver and transmitter.

[6] `https://samy.pl/defcon2015/`

9.3 Attacking Known Radios

There are two very common radios used in IoT, and a few other radios that are specific to the IoT space. We'll cover what these are, how to hack them, and link to more detailed resources where we can.

9.3.1 WiFi Hacking IoT

WiFi hacking is one of the first things most hackers learn these days.[7] We are well used to using `airodump-ng` for logging data and then optimising `aircrack-ng` to crack WPA2 handshakes to get the WiFi key out of it. And yes, all of these apply to IoT/embedded systems and devices.

But, assuming you're already familiar with this stuff, what is specific to IoT? Here are some considerations:

- **DeAuth/DoS Conditions** - Devices do not always maintain connectivity, with or without an attacker nearby causing trouble. So, the question becomes this; *How does the device and/or system deal with outage?*
  - This might not be a security concern on face value, but recent work on the Amazon Key[8] found that if you disable the camera, the stream would display the last

---

[7] To show my age, I remember building a cantenna in high school out of a Pringles can, and soldering it to my very-nice-for-the-time PCMCIA WiFi card to see stuff for *miles* from the top floor of the school. Oh, and warchalking... that was a thing... Good times.

[8] `https://www.wired.com/story/amazon-key-flaw-let-deliv`

    known image... as opposed to some indicator that the service was down. This lead to the Wired article linked below - someone could be there, in view of the camera, but a simple WiFi deauth attack would mean you didn't know.

-

9.3.2 BLE Hacking IoT

9.3.3 ZigBee

9.3.4 LoRa, LoRaWAN, and SigFox

9.4 Attacking an Unknown Radio

To include:

- Using and SDR to record traces in gnuradio
- Using gr-inspector[9] to analyse radio signals
- Using this information to deduce what data is being transmitted
- How to use this to hack the radio interface

[Much more to come as I write it!!]

---

[9] `https://github.com/gnuradio/gr-inspector`

Chapter 10

Putting It All Together

10.1 Background

Now that we have explore the two main basic techniques of accessing UART and dumping firmware over SPI, as well as covering how to analyse firmware and what to look for, we should make plain how this low-level access can inform your pentesting.

I will assume general knowledge, familiarity, and experience with the following types of testing:

- Infrastructure testing - nmap/nessus scanning, and running basic exploits through MSF as a baseline
- Wi-Fi testing - DEAUTH attacks to get handshakes, understanding of WEP/WPA/WPA2, and more recent attacks like Pixie-Dust

- Web Application testing - familiarity with XSS, CSRF, LFI, RCE, SQLi, and the OWASP Top 10 in general
- Mobile Application testing - Testing web API's (XML/-SOAP/JSON) and assessing mobile devices

The reasons for all of these testing methodologies being relevant comes from the vast attack surface presented by embedded devices in a modern world. Many of these were alluded to in the first chapter of Part II, so we will assume that this has been understood for this more practical chapter.

10.2 Hardware Specific Vulnerabilities

Here follows a list of common vulnerabilities that affect only the hardware:

1. UART Access - yes, this should actually be restricted. Many SoC's support the limiting of this service, and it should not be possible to get a route debug shell from accessing the UART on boot.
2. Unsigned firmware images or SecureBoot disabled on ARM SoC's

10.3 Pentesting Embedded Devices - Technical Overview and Attack Models

10.3.1 Web Application

This is perhaps the most fruitful of all the testing areas, so I will mention it first. The first two vulnerability classes are so numerate, that they hvae their own sections.

Remote Command Exec (RCE)

RCE is a high risk, high probability vulnerability on embedded systems, owing to the fact that the main way that performance overheads are mitigated is by the implementation of BASH scripting to perform almost every function on the device. It is not unknown for entire '.asp' websites on embedded systems to be nothing more than renamed BASH scripts that get run on access by the webserver, passing parameters in as you would any other script.

Given the prevelance of this *modus operandi*, it should be tested for exhaustively. Here is a good procedure to gain familiarity with RCE's in embedded systems; Determine whether any user input ends up in any BASH scripts we identified earlier. Common ones include DNS settings, DynDNS settings, NTP timeserver settings, etc. Use the UART connection to test wheter a submitted payload has reached the script unencoded. Once this has been confirmed, different payloads can be created.

It is handy to bear in mind the following RCE payloads that do not use spaces (lines 2,3): (We assume that the UART terminal is mapped to `/dev/ttyS0`)

```
"; echo 'test' > /dev/ttyS0;
";echo$IFS'test'>/dev/ttyS0
";{echo,'test'}>/dev/ttyS0;
```

NB - when testing with `$IFS`, beware when trying certain commands or using pipes. Sometimes you need to add an escape at the end - e.g.

```
nslookup$IFSgoogle.com
```

would need to be

```
nslookup$IFS\google.com
```

on some systems.

Cross Site Reqeust Forgery (CSRF)

CSRF tokens are generated by many embedded platforms, but are almost never checked. The general appearance is that developers of embedded management web-apps appreciate the need for a fix, but fail to understand fully the nature of the vulnerability. Given that most devices will sit on the same local IP address (usually in the 192.168.0.0/16 range) and users almost never change this base address, CSRF has a wide ranging capability. The ability to CSRF a setting/configuration by embedding a javascript payload that talks to the management application has been found to permit the following actions to be performed by payloads loaded over the internet:

- RCE through the local browser
- Setting malicious DNS servers

- Deactivating protection mechanisms (built-in AV and firewalls)
- Uploaded malicious firmware files
- Loading a known stored XSS that then fires to disclose information back to the attacker

(**NB** - this is by no means an exhaustive list, and is included for illustrative purposes.)

Example code to use for testing a payload's effectiveness can be found in the [Appendix I am yet to write].

OWASP Top 10

Taking excerpts of the most significant issues, the following should be kept in mind when pentesting:

Authentication Control and Management

Perhaps the third most common issue on embedded systems, Authentication is often handled in a very lax manner. Things to watch out for include:

- Weak Password Policies or client-side only password strength enforcement
- Passwords not required to be changed after first login to mangement panel
- Access to management screens without authentication (by forced browsing, for example)

- Platform Authentication Bypass - if the system has NTLM or Basic Auth implemented, could this can be bypassed, e.g. by having a valid cookie present

- Predicatable Cookies/Tokens - often, to save on processing and I assume large crypto libraries, hashing is always weak, random numbers generally aren't that random, and so cookies are predominantly time-based and very predictable.

SQLi

Sadly, not much of a thing on embedded devices. Although some do use an sqlite implementation for data storage (very, very few), it is generally not much of an issue owing to a Database server causing significant performance and space overheads that are unnecessary.

It is thereby not something to be overly concerned about if it is not found - a static analysis of the firmware will show you that there probably is no database service to be talking to in any case. **NB** - this does not preclude it being present on externally hosted services!!

Insecure Direct Object Reference (IDOR)

This type of issue is also generally not found on embedded systems, however, their cloud-based counterparts for externally hosted services are often vulnerable, opening up access to other customers' devices in the worst cases.

10.3.2 Infrastructure Vulnerabilities

When pentesting any network service presented on the external WAN or internal LAN/Wi-Fi interfaces there are a number of things you are looking for. Here is how low-level access and firmware analysis can help when assessing this attack surface:

1. Analysis of the bootlog (from UART) will yield precise service versions that nmap/nessus can miss.

2. Analysis of configuration files will yield default passwords or required keys to access and test running services.

3. Analysis of the firmware can determine what SSL Certificates are in play and how they operate (are the private keys universal?).

4. If testing a VOIP service, the VOIP secrets should be analysed to see if they are generic/universal. It is also worth doing SIP testing with tools like `sipvicious` in order to ascertain if there are any issues there.

5. If testing a UPnP service, this should be assessed to see if unauthorized access can be granted through some other (web based) vulnerability (they have been known to be CSRF-able).

6. When testing any serivce, if you successfully generate an exception, then the stack trace has been known to be dumped to serial (in the spirit of debugging). As such, this can be useful for trying/fuzzing payloads to binary or other services on open ports.

10.3.3 Wi-Fi and Wireless testing

Here, we focus on primarily CPE's (home routers or switches with Wi-fi capability), but these can apply more generally to any Wi-Fi interface.

1. Uploading Firmware over Wi-Fi - this should not be possible. Else the attacker who cracks the password and comes back can set a malicious firmware without ever getting physical access to the device.
2. It should not be possible to determine the default Wi-Fi password or WPS pin by sniffing for publically available information (such as the default SSID or MAC address)
3. It should not be possible to bruteforce WPS pins nor trigger a pairing without physical access to the device.
4. Attacks that are more recent, such as the Pixie-Dust attack I mentioned earlier, should be guarded against, as well. This is unlikely in many cases, owing to the lack of updating base software in the firmware.

See the next chapter for more information about WiFi, BLE, and other radios in use on embedded/IoT systems.

10.3.4 Mobile App/Cloud Integration

We assume for this testing that there is a tie-in mobile app, or management mobile app that may also have some 'cloud'-based connectivity.

1. Many apps permit management of the device itself - and the credentials to allow this access should not be pushed to `adb logcat`. This should be checked during standard operation.

2. The cloud-based service should be checked for all the usual problems, with special attention paid to username enumeration and bruteforcing - such a compromise can lead to actual compromise of real devices.

3. If the cloud-based platform or mobile application permits the ‘adding’ of devices, it should be confirmed that no Insecure Direct Object References (IDOR) vulnreabilities could be used to add devices arbitrarily.

4. Tying into RCE vulnerabilities, if any are identified locally, then these should be tested to see if they can be chained with a CSRF or over the app API itself to trigger such a vulnerability over the internet.

We’ll cover more details regarding mobile app testing for IoT and cloud hosting tests relative to IoT.

10.3.5 Denial of Service (DoS)

This can be brought about in a myriad of ways, and it is very hard to give general guidance. However, the following examples have been found in the wild:

- Bad Error Handling

- Unexpectedly large parameters on web headers (such as User-Agent or Referer)
- Incorrect Basic Auth headers - where the header is invalid Base64 has crashed a device completely
- TR069 Port on external interface - DoS through fuzzing.

In more general terms, it should be assessed wherever some level of instability of processing is found in the system.

10.3.6 Additional Testing Points

Bluetooth/BLE

Many BLE chips have custom debug stacks that are now well known, and tend to run their own firmware. As such, it should be seen if the devices can be connected to without any keys, or with default/universal keys. They should be segregated from the rest of the device's functionality, and this should be confirmed (it shouldn't be possible to cause a system-level exception from the BLE data being sent to the SoC).

See the chapter on radio hacking for more information about BLE hacks.

Part III

Advanced Testing Topics

Ideas include:

- running firmware files in virtual machines (qemu)
- JTAG/SWD in practice
- Glitching
- IC Decapping - with and without acids - and silicon die reverse engineering
- Reverse Engineering bootloaders
- worked example of bit-banging a protocol - CC2541 custom dbg??
- CRP Bypasses

Chapter 11

Virtualizing Firmware

11.1 The Problem

Devices are fickle and delicate and prone to, oh, blowing up or dying or not working anymore, or even not being provided by a client in time for an assessment, sometimes. This is annoying, and hard to deal with, but part of the many facts of life.

However, we have virtual machines, and indeed, virtual appliances (such as virtual firewalls appliances, etc.) - these are just some virtual machines that run the firmware for the desired security appliance in a sandbox/VM/container. Additionally, it is well known that QEMU, for example, supports emulation for ARM/MIPS/x86/etc., and makes them available for access on an x86 machine.

Wait, so... why can't we virtualise firmwares then?

Well, the fact of the matter is, *you can*... -ish... kinda... permit me to explain.

11.2 The Solutions

As we have seen earlier, there are fundamentally two main types of firmware - those running a Linux-like environment, or those running straight code. We'll run through how you can virtualise both - with examples linked, but more importantly, we will aim to include enough theory that you can take something that hasn't been virtualised before and actually have a reasonable chance of achieving this.

11.2.1 Emulating Linux-based Router Firmware with Firmadyne

Firmadyne can be cloned from the firmadyne repo: `https://github.com/firmadyne/firmadyne` ← You'll need this to understand this part of the chapter.

Firmadyne was developed to perform the following actions automatically:

1. Search vendor websites for firmware, and download them
2. Unpack the firmware and extract the filesystem.
3. Virtualise this in qemu for either MIPS or ARM as needed.
4. Run exploits/nmap scripts/metasploit modules against the running firmware.

However, for our purposes, it will serve to be able to run some various firmwares virtually, meaning we can do auxiliary/automated testing of our own, without needing to sacrifice the device or restrict our access to it.

The workflow for firmadyne internally is as follows:

1. Use `scraper` to find files on the vendor site and download them.
2. Let `extractor` extract the relevant binary data from the firmware file.
3. Setup `qemu` using scripts and two custom binaries:
    - `libnvram` - an NVRAM peripheral emulator compatible with qemu
    - `console` - a serial console device emulator compatible with qemu
4. Choose the appropriate architecture and run the firmware virtually
5. Invoke custom and metasploit based exploit scripts, and run nmap scans on the running target

We wish to essentially use the automatic part of this process for steps 3 and after in our work, having done the first two steps manually before now.

Roughly, we would do this as follows:

1. Extract the firmware and filesystem as we have shown previously - judicious use of `binwalk`, `unsquashfs`, `jefferson` , and other tools will make this possible and reasonably straightforward.

2. Perpare the firmware extracts to run in qemu; initialise the filesystem, and setup the NVRAM emulator and console device.

3. Run the relevant QEMU script in the firmadyne repo.

With a lot of luck, it should now be running, however, usually you can expect to have to fix the following things:

- Memory mappings for UART devices
- Memory mappings for network devices
- Memory mappings for GPIOs
- Locating the entrypoint for the firmware's kernel

NB - most of these can be fixed with some patience, some thought, and some steady immersion in the relevant data sheet for the SoC this is supposed to be running on (if it is available).

11.2.2 Emulating STM32 Firmware with `qemu-stm32`

Anyone who has ever fought with GRUB2 or a Linux ditribution that doesn't have a binary in its package manager (or even not having a package manager[1]) - Linux internals are scary for beginner, but there are enough people around who have solved the problem (and posted it to StackOverflow) for us to not be too worried.

But what about those pesky firmwares that *just run code*??!!

---

[1] I leave it for history to decide whether my love of Slackware was the result of Stockholm Syndrome or not...

Well, for the STM32 there is the `qemu-stm32` fork of QEMU that permits the ARM-SoftMMU to run an STM32 firmware, and even emulate parts of a development board for it!

You can download qemu-STM32 from the repo here: `https://github.com/beckus/qemu_stm32`

For a walkthough and tutorial for how to use these see here: `https://github.com/beckus/stm32_p103_demos`

[More details to come... would be nice to show how to take a memory map and implement that for virtualisation]

Chapter 12

Practical JTAG/SWD Debugging

12.1 What is JTAG and SWD?

12.1.1 Motivation

We saw JTAG and SWD before in this volume - in 3.4, for reference. However, in order to really get to grips with hardware hacking, we need to dive in-depth and explore the protocols in full. So, let's go!

JTAG, or 'Joint Test Action Group'[1] is a consortium of manufacturers who defined the standard that we now refer to by the same name. *But what is it??*

Quite simply, it's a debug and testing protocol designed to add various troubleshooting and programming features to IC's of any package. It was envisioned that such a thing was required with the advent of BGA packages in particular. Why BGA? Well, principally it comes down to soldering errors. With packages with external pins - like LQFP for example - it's easy to either visually or physically test it with probes. With a BGA you can't see under the component, and you can't access it with probes - so what do you do? You use something like JTAG. Hopefully we'll explain how in the next section.

---

[1] Resist the urge for 'legal state quality assurance' jokes, please...

This said, the functionality of JTAG expanded significantly, so we need to talk about JTAG functions, the state machine, and how it ties together. We will also discuss Serial Wire Debug (SWD) which is an implementation of JTAG used by ARM-based processors. For the purposes of practical action, they are the same, so we will treat them as such (and highlight any differences as and when they arise)[2]. As such, when I say JTAG, you can substitute SWD, and vice versa, without any loss of generality.

12.1.2 JTAG/SWD Functions

There are three principle functions performed by JTAG/SWD:

1. Loading Firmware
2. Debugging the system
3. Boundary Scan Testing

---

[2] The ARM Debug Interface Architecure v5 is the definitional point for SWD - it can be found here: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031c/index.html`

We'll cover these in reverse order, especially as people are probably not very familiar with the term 'boundary scan'.

Boundary Scan:

JTAG has the boundary scan silicon literally all around the boundary of the silicon die of an IC. As such, it can talk to every pin directly, and set its level to high or low, as well as read the level of every pin. Kinda neat! Very useful for hacking, as I'm sure your brain is thinking right now.

The boundary scan is when the JTAG state machine sends a binary string around all the pins, set the boundary nodes to read, and sees how the string differs on return. What use is that? Well, suppose we send out

$$00000000000000000000000001000000$$

- to set pin 26 to high - and we get back

$$00000000000000000000000001100000$$

- with both pins 26 and 27 high. This would indicate a possible solder short or some other physical issue with the board.[3]

Of course, we can also use this to manipulate the operation of the chip or read pin values as we need during regular operation - it has even been shown that you can emulate UART over JTAG, but this is non-trivial and very unreliable compared to the easier methods you could employ.

---

[3] Yes, this is a slightly over-simplified and maybe contrived example, but it illustrates the point and one of the original intended uses of JTAG.

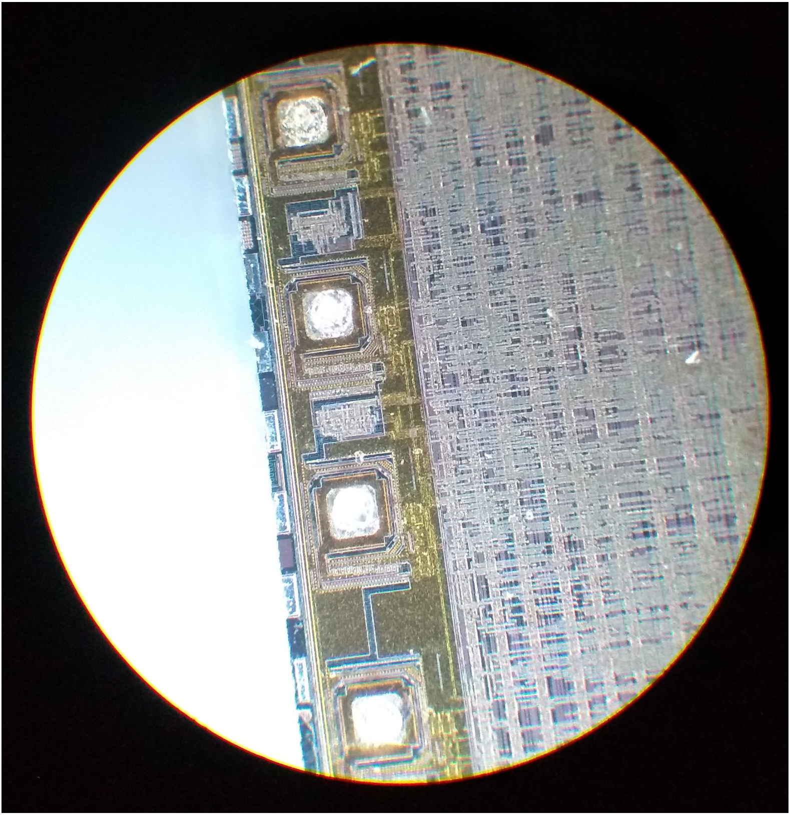


Figure 12.1: Showing the boundary modules next to pins on a silicon die under a microscope

Debugging:

JTAG was also designed to allow a developer to debug the

CPU and periperals[4] onboard the microcontroller. As you would expect, you can perform all the usual debugging operations:

- Setting breakpoints
- Halting the CPU
- Setting register values/manipulating the $PC
- Dumping memory contents (for disassembly, say)

OpenOCD as part of its normal operation will spawn a GDB server that will happily interface GDB to our target with ease (see a worked example later). Thus, you can reduce the problem of debugging a chip quite easily to debugging any binary in the familiar GDB context.

Loading Firmware:

JTAG can load firmware on, and off, the microcontroller. Yes, you read that right. You can often retrieve firware using JTAG/SWD on a target IC. This will, of course, depend on the settings of the security bits on the flash (See the chapter on CRP bypass for how these can sometimes be circumvented).

OpenOCD has, for many microcontroller architectures, implemented flash download and flash upload functionality, and this is the same for other software packages for specific debuggers, such as J-Link.

---

[4] Recall that SPI or CAN or UART interfaces are all ‘peripherals’ included in the microcontroller silicon - they really are just placed and wired up in a similar fashion to peripherals in macro computers.

12.2 Fully worked JTAG/SWD example

The road to JTAG/SWD is paved with good intended features.

Here we will describe the process of getting SWD on a STM32F103 chip by means of a STLink-V2 - specifically, a cheap chinese knock off, and a 'blue pill' development board as our target.

First we wire up the devices. Here are the pin mappings:

- SWDIO goes to SWDIO
- SWDCLK goes to SWDCLK
- GND goes to GND
- VCC goes to VCC - assuming we want to power the chip through the debugger
  - If the IC gets power elsehwere, don't connect this.
  - If connecting to a target currently *in situ*, double check for any ground loops before powering anything.

We will use OpenOCD - we cover the installation and configuration of this in 3.4, so we will assume that this is installed.

First off, we run this command:

```
sudo openocd -f interface/MyConfig.cfg -f target/
    stm32f1x.cfg
```

This will run the OpenOCD servers in the background. In order to pull the firmware, we will need to connect to the telnet server using this command:

```
telnet localhost 4444
```

[Maybe put some stuff in here about ELF-ing a firmware for STM32? It's a bit niche, though...]

Once we want to start debugging, we can open the appropriate GDB (such as `arm-none-eabi-gdb`) and connect to the local GDB server that will interface us to the IC as follows:

```
target remote localhost:3333
```

Once we are all connected we are ready to begin debugging our target as if it were any other binary process we are familiar with debugging. Success!!

12.3 JTAG Disabled Bypass

(Material in this section is based on `https://medium.com/@LargeCardinal/how-to-bypass-debug-disabling-and-crp-on-stm3`

12.3.1 Overview

There is a good case to state that modern devices are moving towards a very slightly more homogenised ecosystem. This is signalled by the number of cheap, easily available ARM-based microcontrollers and microprocessors available on the market.

ARM offers many features 'as standard' across various applications, and with low-price IC's on the market, there is a good chance that the uptake of ARM architecture will continue. The

STM32 series of microcontrollers are quite popular with developers owing to good physical/performance properties, as well as a low price point.

We will consider the STM32F103 - specifically the STM32F103C8T6 IC - and how you can disable debug protections and CRP. These are nifty little Cortex-M3 baesd chips with 64kb flash, 20kb RAM, and clocked up to 72MHz. You can see, perhaps, why they might be popular!

12.3.2 The Problem

Implementations of ARM can vary wildly. There are some standard features that are either missed out or reconfigured completely - each IC manufacturer is left to their own devices (→pun) for how they produce ARM chips.

There are two protection mechanisms that are usually recommended by hardware/embedded/IoT security people:

1. Disable Debug ports - JTAG and SWD disable
2. Enable firmware protections e.g. Code Readout Protection or CRP

We want to look to see if we can disable these protections on the target IC - the STM32F103.

12.3.3 Disabling Debug

The first is quite straightforward - for STM32F103 you simply set `__HAL_AFIO_REMAP_SWJ_DISABLE();` in the standard file `stm32f1xx_hal_msp.c` (if you are using STCubeMX generated output - you can do something similar with libopencm3, too).

This is the default configuration of STCubeMX-generated firmware source projects. Meanwhile, the IC's themselves ship with SWD Enabled but the flash being locked - but this is trivial to unlock over SWD.

But once such a firmware with SWD/JTAG debugging disabled is applied, then this device no longer can be debugged. At least, that is the idea.

So, our initial scenario is an STM32F103 device with Debug Disabled.

12.3.4 Bypassing Disabled Debug

All this said, there are two pins that should be brought to attention - the BOOT0 and BOOT1 pins. These are designed to be used to trigger fallback/default modes on the IC - and as usual, this can spell disaster if you're trying to hide some IP.

Specifically, we first want to use BOOT0, which is pin 44 on the LQFP48/UFQFP48 packages. Here is the attack for this:

12.3.5 Attack Outline

- We need to raise this BOOT0 pin to 3.3V and then power cycle the IC
- Acquire a copy of the firmware
- Patch the firmware to enable SWD debugging

- Re-flash the new firmware to have the device with debugging enabled

So, first, we raise the voltage of pin 44 to 3.3V. If you're playing along at home with a Blue Pill development board, you'll note that this is one of the two jumper pin options on the top of the board.

Here is the board before and after we apply the change to BOOT0:

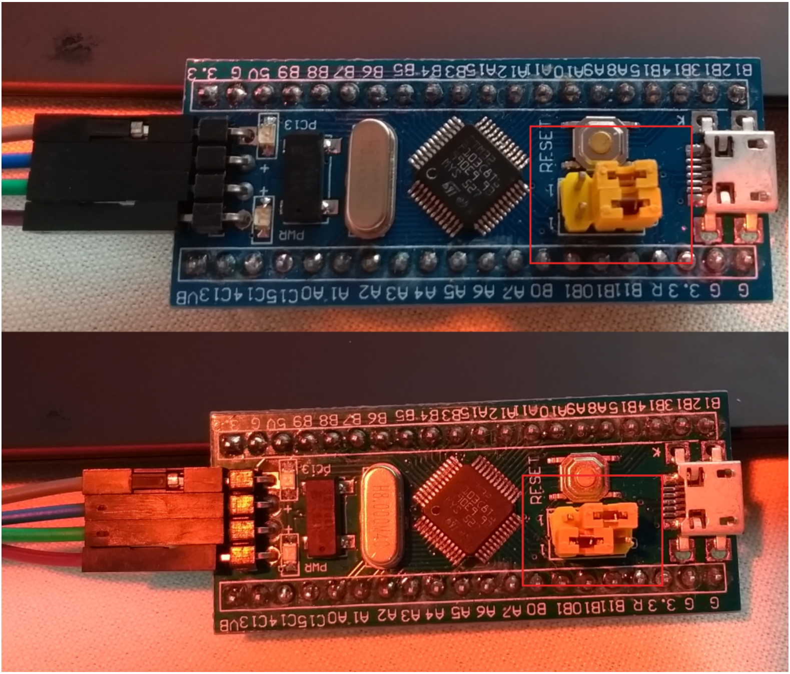
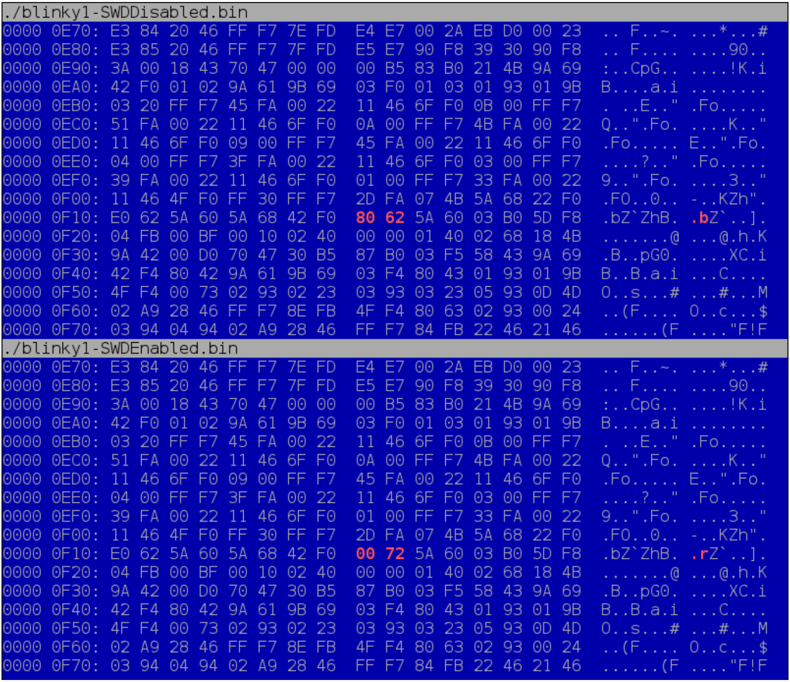


Figure 12.2: Showing the change of pin.

We will now see that we can debug the IC, and dump firmware using SWD. That's it. That's the first part of the attack. Having issues? You can just use mbr to cycle through all the addresses on the flash using a script, or dump the image, both using the openOCD client, an STLink-V2 USB-to-SWD interface, and a little bit of time/SSD space.

Once you have acquired the firmware, how can we enable debugging on the code? Well, first, we reassemble the firmware into some `firmware.bin` file.

With this done, we need to do some binary editing. Here is the diff between two otherwise identical firmware files - the first has debug disabled, and the second (lower) has SWD Enabled.



```
./blinky1-SWDDisabled.bin
0000 0E70: E3 84 20 46 FF F7 7E FD  E4 E7 00 2A EB D0 00 23  .. F..~. ...*...#
0000 0E80: E3 85 20 46 FF F7 7F FD  E5 E7 90 F8 39 30 90 F8  .. F.... ....90..
0000 0E90: 3A 00 18 43 70 47 00 00  00 B5 83 B0 21 4B 9A 69  :..CpG.. ....!K.i
0000 0EA0: 42 F0 01 02 9A 61 9B 69  03 F0 01 03 01 93 01 9B  B....a.i ........
0000 0EB0: 03 20 FF F7 45 FA 00 22  11 46 6F F0 0B 00 FF F7  . ..E.." .Fo.....
0000 0EC0: 51 FA 00 22 11 46 6F F0  0A 00 FF F7 4B FA 00 22  Q..".Fo. ....K.."
0000 0ED0: 11 46 6F F0 09 00 FF F7  45 FA 00 22 11 46 6F F0  .Fo..... E..".Fo.
0000 0EE0: 04 00 FF F7 3F FA 00 22  11 46 6F F0 03 00 FF F7  ....?.." .Fo.....
0000 0EF0: 39 FA 00 22 11 46 6F F0  01 00 FF F7 33 FA 00 22  9..".Fo. ....3.."
0000 0F00: 11 46 4F F0 FF 30 FF F7  2D FA 07 4B 5A 68 22 F0  .FO..0.. -..KZh".
0000 0F10: E0 62 5A 60 5A 68 42 F0  80 62 5A 60 03 B0 5D F8  .bZ`ZhB. .bZ`..].
0000 0F20: 04 FB 00 BF 00 10 02 40  00 00 01 40 02 68 18 4B  .......@ ...@.h.K
0000 0F30: 9A 42 00 D0 70 47 30 B5  87 B0 03 F5 58 43 9A 69  .B..pG0. ....XC.i
0000 0F40: 42 F4 80 42 9A 61 9B 69  03 F4 80 43 01 93 01 9B  B..B.a.i ...C....
0000 0F50: 4F F4 00 73 02 93 02 23  03 93 03 23 05 93 0D 4D  O..s...# ...#...M
0000 0F60: 02 A9 28 46 FF F7 8E FB  4F F4 80 63 02 93 00 24  ..(F.... O..c...$
0000 0F70: 03 94 04 94 02 A9 28 46  FF F7 84 FB 22 46 21 46  ......(F ...."F!F
./blinky1-SWDEnabled.bin
0000 0E70: E3 84 20 46 FF F7 7E FD  E4 E7 00 2A EB D0 00 23  .. F..~. ...*...#
0000 0E80: E3 85 20 46 FF F7 7F FD  E5 E7 90 F8 39 30 90 F8  .. F.... ....90..
0000 0E90: 3A 00 18 43 70 47 00 00  00 B5 83 B0 21 4B 9A 69  :..CpG.. ....!K.i
0000 0EA0: 42 F0 01 02 9A 61 9B 69  03 F0 01 03 01 93 01 9B  B....a.i ........
0000 0EB0: 03 20 FF F7 45 FA 00 22  11 46 6F F0 0B 00 FF F7  . ..E.." .Fo.....
0000 0EC0: 51 FA 00 22 11 46 6F F0  0A 00 FF F7 4B FA 00 22  Q..".Fo. ....K.."
0000 0ED0: 11 46 6F F0 09 00 FF F7  45 FA 00 22 11 46 6F F0  .Fo..... E..".Fo.
0000 0EE0: 04 00 FF F7 3F FA 00 22  11 46 6F F0 03 00 FF F7  ....?.." .Fo.....
0000 0EF0: 39 FA 00 22 11 46 6F F0  01 00 FF F7 33 FA 00 22  9..".Fo. ....3.."
0000 0F00: 11 46 4F F0 FF 30 FF F7  2D FA 07 4B 5A 68 22 F0  .FO..0.. -..KZh".
0000 0F10: E0 62 5A 60 5A 68 42 F0  00 72 5A 60 03 B0 5D F8  .bZ`ZhB. .rZ`..].
0000 0F20: 04 FB 00 BF 00 10 02 40  00 00 01 40 02 68 18 4B  .......@ ...@.h.K
0000 0F30: 9A 42 00 D0 70 47 30 B5  87 B0 03 F5 58 43 9A 69  .B..pG0. ....XC.i
0000 0F40: 42 F4 80 42 9A 61 9B 69  03 F4 80 43 01 93 01 9B  B..B.a.i ...C....
0000 0F50: 4F F4 00 73 02 93 02 23  03 93 03 23 05 93 0D 4D  O..s...# ...#...M
0000 0F60: 02 A9 28 46 FF F7 8E FB  4F F4 80 63 02 93 00 24  ..(F.... O..c...$
0000 0F70: 03 94 04 94 02 A9 28 46  FF F7 84 FB 22 46 21 46  ......(F ...."F!F
```

Figure 12.3: vbindiff view - look at the bits in RED

As you can see, at 0x00000F18, the 2-byte word there is the only thing that is different. So, search for 0x8062 at the same address in our target firmware and patch it to be 0x0072, and then re-flash the firmware onto the STM32F103 chip.

We can then lower the BOOT0 pin voltage, power cycle the IC, and then we should have (possibly with a little more hacking) SWD enabled on our target firmware.

12.3.6 Remarks

Although quite simple, this is a valid attack when you consider that

$$\text{‘disabling debug’} == \text{‘firmware protection’}$$

in the minds of some developers and frameworks. As such, attacks like this both help a researcher/assessor potentially access a chip, and demonstrate by Proof of Concept[5] that this alone is not an effective countermeasure.

See Chapter 18 for details on other hardware control bypasses on firmware!

---

[5] Remember: PoC——GTFO.

Chapter 13

Glitch Attacks - Real World Fault Injection

13.1

13.1.1

Chapter 14

Side-Channel Attacks

14.1

14.1.1

Chapter 15

IC Decapping

15.1

15.1.1

Chapter 16

Bootloader Reverse Engineering

16.1

16.1.1

Chapter 17

Custom Protocol Implementations

17.1 How do you solve a problem like a custom protocol?

By and large, in the modern technological world, we find that most devices will use common protocols - the ones you have seen covered in previous chapters.

However, it is not uncommon to see some custom protocol implemented - whether it be some debug or custom method transferring serial data, we will present some ideas around how these can be reverse engineered, and then bit-banged on some uC we have lying around.

17.1.1 Bit-Banging - a beginner's guide

'Bit-banging' may sound like something from a *Gibson-esque cybersexual noir* book, but in actual fact, it's a way of using GPIO pins in order to emulate any protocol we want.

Similar to when we defined a radio in software and then used an SDR to then transmit and receive, GPIO pins are flexible ways to read digital levels and output digital levels. As is hopefully reasonably apparent by now, many serial protocols work on measuring voltages and comparing against a threshold, and being able to set a line to be some particular voltage.

By creative use of internal clocking and setting GPIO pins to appropriate levels, and then reading these levels, we can effectively emulate a protocol without having a specific protocol silicon peripheral on our microcontroller.[1]

17.1.2 Basic Example - UART Rx

So, for a first example we're going to big-bang a UART receiver on an Arudino Nano (though this will work on any arduino). We're going to keep it simple, and here's the rough design; we'll listen for a UART byte being transmitted, receive it, and then send it out the regular UART so we can see it.

Let's recall what happens during a regular (non-inverted) UART transmission[2]:

---

[1] After all, these are just doing the same as our software would do, but hard-coded into silicon hardware.

[2] We covered this earlier in a timing diagram, but we'll describe it here in words.

1. We will assume a baud rate of 9600, with a setup of 8N1.
2. The line is held high until a stop bit is sent by holding the line low.
3. Once we see this, we then wait 104 microseconds (the time it takes for one bit on 9600 baud)
4. Next we enter a for loop where we see if the line is high, and if so, put a ‘1’ in the right position in the byte
    - If the line is held low, we will do nothing, effectively recording a zero.
5. We then print what we saw out the regular UART so we can check.

We will work that we connect the grounds on the transmitter and receiving arduino, and will connect the Tx of the UART source to pin D2 on our arduino. The code that accomplishes this task is as follows: (**Note** - this will compile in Arduino IDE for pretty much any arduino)

```
// get delay time for 9600 baud UART
// take 1/9600 and this will give you the time in microseconds
int DELAY = 104;

// setup char for UART
char a;

void setup() {
  pinMode(2, INPUT); //setup pin as input
  Serial.begin(9600);
}

void loop() {
  if (!digitalRead(2)) {
    // we have a start bit!
    // so, wait for the first bit...
    delayMicroseconds(DELAY);
    for (int i = 0; i < 8; i++) {
      // check if the line is high...
      if (digitalRead(2)) {
        // if so, put a one in the correct position
        // remembering the transmission order of the bits
        a += (1 << i);
      }
      // if not high, then leave it as 0 and wait...
      delayMicroseconds(DELAY);
    }
  // print the byte we got...
  Serial.print("We got: ");Serial.println(a);
  // reset a to null...
  a = NULL;
  }
// let's go round again!
}
```

We can follow a similar procedure for UART transmit, but we leave this as an exercise for the reader.

This is a very straightforward example of how to bit-bang a protocol, and as such, hope that this example goes some way to demystify what we are doing.

Protocols that are relatively easy to bitbang include SPI, I^{2}C, and debugging protocols such as JTAG/SWD not being particularly more complicated once you understand the workings of the internal state machines.

17.2 Implmenting a Custom Protocol w/ Datasheet

Here we will show how we can build a protocol library for an aruidno by making use of the excellent CCLib by Ioannis Charalampidis `https://github.com/wavesoft/CCLib` - an arduino implementation of the Texas Instruments CC Debugger protocol. Handy to use with BLE chips such as the CC2541 or CC11xx series.[3]

To do this, we will reference one of TI's own documents about implementing this protocol, with examples drawn from the CCLib code as appropriate: `http://www.ti.com/lit/an/swra410/swra410.pdf`. We also cite the debug protocol as another useful reference: `http://www.ti.com/lit/ug/swra124/swra124.pdf`

17.2.1 Analysing the Protocol Wiring

Many protocols have reasonably straight forward wiring, but understanding how these things can interact can be helpful in deciphering what we are seeing on an oscilloscope. The reference document gives us three pins required by CC debug protocol:

---

[3] Historical note: CC stands for 'ChipCon', a company that made lots of low-power IC's for short-range radio. They were acquired by TI in 2006 for around $200 million. Although they are owned entirely by TI, they still seem to operate as a separate unit, releasing chips with their trademark "CC" prefix, and making use of their debug protocol.

- `RESET_N` - Reset line, that is required by the protocol.
- `DC` - Debug Clock
- `DD` - Debug Data

Note, things to keep in mind when analysing the wiring for any given protocl include:

- Whether the protocol uses pull-up or pull-down resistors.
- Whether the protocol uses inverted logic (or some hybrid, like CAN bus).
- Whether any lines are redundant - such as the reset lines on JTAG/SWD.
- Whether any usually-redundant lines are actually useful - such as DTR and DSR for full RS232.

Note that CCLib splits the `DD` line in two - it has one pin acting as input, `CC_DD_I`, and one acting as output, `CC_DD_O` - this saves time on the arduino switching modes on a single pin. The wiring for this is a fairly common hack; a 100kΩ resistor is placed between `CC_DD_O` and `CC_DD_I` (with another 200kΩ resistor going from this connection to ground), with the actual `DD` line connected to `CC_DD_I`. Similar 100kΩ resistors are placed infront of the other output lines.

This allows us to work more easily with the signals and processing on the slightly limited arduino.

17.2.2 Analysing The Protocol Timings

As we hope is clear from the UART Rx bit-bang example, much of this requires an understanding of the timings - what lines need to be raised or lowered, and when. Important aspects of timings when working out how to implement a protocol for use include:

- Making sure that the line is held high in time for the target to read a rising clock edge (or falling edge, if this is what is used)
- Dealing with accumulating timing errors for continuous streams
  - In the above example, we should have used 104.1666... as our microsecond delay.
  - The 0.1666... error will accumulate, so we will lose a microsecond every 6 bits or so.
  - This is fine for the 10 bits we are concerned with for each UART transmission, but other protocols are much more sensitive to timing errors
- Protocols such as CAN bus that have sometimes confusing artefacts, like bit-stuffing, as part of the protocol.

However, let's consider the timing requirements for the CC Debugger protocol we are concerned with here.

- The clock signal drives all of the communications - including the `DEBUG_INIT` start.
- Data is driven on `DD` by the rising/positive edge of the `DC` line.

17.2.3 Using GPIOs to Implment a Protocol

This is the bit we have been building up to - how to use the information we have gathered and then create an implementation that works for what we need.

It is worth keeping in mind, CCLib is quite a mature library - it has a python interface to communicate to the Arduino, and has implemented error logging/checking and a custom protocol to signal that back to the python interface and alert the user. This is quite a lot of work - and sometimes it will be required (for which CCLib is a good example).

However, some situations will only require that you implment one function in order to prove a PoC, hook in a fuzzer, or create a manual testing interface you can use in security assessment of a device.

We will not give a full teardown of the CCLib implementation - we will, instead, use three illustrative examples; `DEBUG_INIT()`, `GET_CHIP_ID()`, and `CHIP_ERASE()`.

All examples are taken from `CCDebugger.cpp` from the CCLib library, in the Arduino/CCLib folder.

`DEBUG_INIT()` Implementation

Kicking a CCxxxx target into debug is fairly straightfoward:[4]

1. Pull `RESET_N` low

---

[4] See page 6 of the TI reference; `http://www.ti.com/lit/an/swra410/swra410.pdf`

2. Pump `DC` twice[5]

3. Pull `RESET_N` high

This is fairly straight forward in code:

```
/**
 * Enter debug mode
 */
byte CCDebugger::enter()
{
  // ------- SNIP -------//

  // Reset error flag
  errorFlag = CC_ERROR_NONE;

  // Enter debug mode
  digitalWrite(pinRST, LOW);
  cc_delay(200);
  digitalWrite(pinDC, HIGH);
  cc_delay(3);
  digitalWrite(pinDC, LOW);
  cc_delay(3);
  digitalWrite(pinDC, HIGH);
  cc_delay(3);
  digitalWrite(pinDC, LOW);
  cc_delay(200);
  digitalWrite(pinRST, HIGH);
  cc_delay(200);

  // We are now in debug mode
  inDebugMode = 1;
```

---

[5] In Travis Goodspeed's debug interface for CC430 in the GoodWatch project, this line is pumped a few times more, just to be sure.

`GET_CHIP_ID()` Implementation

Getting the Chip ID is a useful way of seeing how data transfer works. First, we will summarise some of the utility functions used in this part of the CCLib code:

- `read()` - this function sends 8 clock pulses, and reads `DD` line for incoming data. Here is a code snippet

```
  // Send 8 clock pulses if we are HIGH
  for (cnt = 8; cnt; cnt--) {
    digitalWrite(pinDC, HIGH);
    cc_delay(2);

    // Shift and read
    data <<= 1;
    if (digitalRead(pinDD_I) == HIGH)
      data |= 0x01;

    digitalWrite(pinDC, LOW);
cc_delay(2);
```

- `write()` - This does the inverse action, sending 8 clock pulses and then raising or lowering the `DD` line according to the byte we want to send. Here's a snippet:

```
  // Sent bytes
  for (cnt = 8; cnt; cnt--) {

    // First put data bit on bus
    if (data & 0x80)
      digitalWrite(pinDD_O, HIGH);
    else
      digitalWrite(pinDD_O, LOW);
    // Place clock on high (other end reads data)
    digitalWrite(pinDC, HIGH);

    // Shift & Delay
    data <<= 1;
    cc_delay(2);

    // Place clock down
    digitalWrite(pinDC, LOW);
    cc_delay(2);
}
```

- `switchRead()` and `switchWrite()` - these helper functions let the code switch modes between reading and writing data to the `DD` line.

- The code has the ‘read chip ID’ command, `0x68`, in a variable we can call.

Here is a snippet of the relevant code:

```
write( instr[I_GET_CHIP_ID] ); // GET_CHIP_ID
switchRead();
bRes = read(); // High order
bAns = bRes << 8;
bRes = read(); // Low order
bAns |= bRes;
switchWrite();
```

We can read the response, and then get an idea of which chip is in play. For example, a response of `0xA5` would indicate a CC2530 target, whilst `0xB5` would indicate a CC2531. See page 7 of the technical reference.[6]

---

[6] `http://www.ti.com/lit/an/swra410/swra410.pdf`

This example shows very well how we can bit-bang a custom protocol in software to both send data, and read the responses back. It is worth noting that the CCLib code has other commands available, not just `GET_CHIP_ID`, such as:

```
    // Default CCDebug instruction set for CC254x
    instr[INSTR_VERSION]     = 1;
    instr[I_HALT]            = 0x40;
    instr[I_RESUME]          = 0x48;
    instr[I_RD_CONFIG]       = 0x20;
    instr[I_WR_CONFIG]       = 0x18;
    instr[I_DEBUG_INSTR_1]   = 0x51;
    instr[I_DEBUG_INSTR_2]   = 0x52;
    instr[I_DEBUG_INSTR_3]   = 0x53;
    instr[I_GET_CHIP_ID]     = 0x68;
    instr[I_GET_PC]          = 0x28;
    instr[I_READ_STATUS]     = 0x30;
    instr[I_STEP_INSTR]      = 0x58;
instr[I_CHIP_ERASE] = 0x10;
```

`CHIP_ERASE()` Implementation

This is the last in our basic examples of implementing the CC Debug protocol in software on an arduino.

The `CHIP_ERASE()` function has a slightly different operation than the read/write we have seen so far:

1. Send the command `CHIP_ERASE()` by writing the value `0x10` to the target
2. Wait for the `DD` line to go low.
3. Send 8 clock pulses and check if the 7th bit is high - if so, this means `CHIP_ERASE_BUSY`

4. Once this is unset, the flash is erased

It should be noted that the third and fourth steps in this description are not observed in the CCLib code strictly. The command `CHIP_ERASE()` is sent to the device and then the status is read back to the user immediately:

```
  byte bAns;

  write( instr[I_CHIP_ERASE] ); // CHIP_ERASE
  switchRead();
  bAns = read(); // Debug status
  switchWrite();

return bAns;
```

As such, an understanding of how this code works may help in troubleshooting such code, should a mass erase operation fail.

17.3 Reverse Engineering and Implementing a Custom Protocol

[Find an example IRL of a custom protocol. Write up should be - RE it, and then bit bang on an ARM or arduino uC...]

Chapter 18

CRP Bypass Techniques

18.1 Overview of CRP

18.1.1 What is CRP?

CRP, or Code Readout Protection, is a collection of various methods employed by IC manufacturers to ensure that access to the firmware loaded onto a chip is limited once the chip has been flashed.

The intention is to prevent the acquisition of firmware by unauthorized parites - this is more to prevent copying of devices and IP[1]/copyright/patent infringement. However, as we shall see, there are a variety of ways in which CRP has been found to be bypassed.

---

[1] Intellectual Property, not the other one.

18.1.2 CRP Bypasses - the Basics

CRP is generally only applied to microcontrollers - and is not found on microprocessors, where the RAM and ROM are in separate packages. As such, you can't really control the firmware as the ROM IC is open for an interested party to access and pull off the firmware (as has been covered earlier).

CRP is a control specific to microcontrollers - and as such, there are as many ways of implementing it as there are manufacturers of uC's! There are broadly several 'levels' of CRP:

- No control applied
- Protect from read, but allow erase/rewrite
- Protect from read, disallow erasure

As such, we want to achieve one of the following situations:

- Disable the security control altogether, giving free access to the ROM
- Find some other way of accessing the information through another channel, say from RAM
- Leverage fail-safe or fail-open modes on IC's

We shall cover Glitch-based firmware recovery or CRP bypass in Chapter 13, so please see there for details. This chapter will assume that you have access to the target device, and suitable equipment for gaining debug access - UART interfaces, JTAG/SWD adapters, etc.

This chapter is broadly influenced and inspired by Andrew Tierney's[2] talk given at BSides Leeds entitled 'Armadillos!'[3]

18.2 CRP Bypasses

18.2.1 UV Light to Reset Security Fuses

This attack was formulated by Bunnie Huang for opening up firmware on a locked PIC 18F1320 - Source: `http://www.bunniestudios.com/blog/?page_id=40`

How UV affects storage

In order to understand this attack we need to go down to the very small and understand what a transistor looks like under a microscope.

Figure 18.1 shows the structure of a floating gate MOSFET (FGMOS) - these use a neat effect called Fowler-Nordheim tunnelling[4] to 'store' a charge in the floating gate.

There are two ways in which this information can be reset - by either applying a large inverse voltage (which at these scales is in the 3-5V range) or by applying UV light (typically c. 205nm wavelength is used).

---

[2] `https://twitter.com/cybergibbons`

[3] A recording can be found here: `https://www.youtube.com/watch?v=DTuzuaiQL_Q`

[4] I tell ya, this stuff is magical physics...

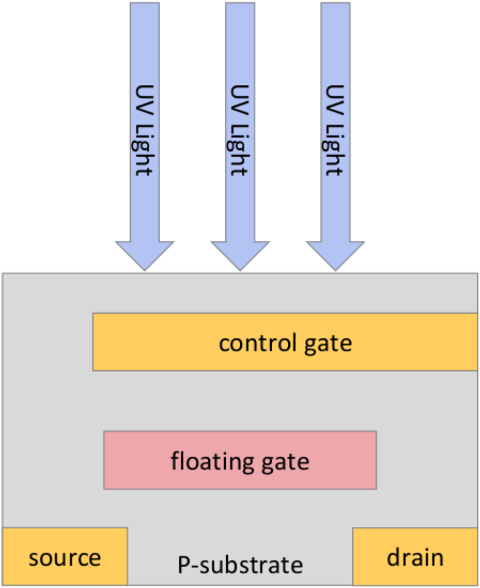

UV Light
UV Light
UV Light
control gate
floating gate
source
P-substrate
drain

Figure 18.1: Structure of a FGMOS - the UV light resets the state of the floating gate

To make this a viable way of resetting, say, security bits that prevent us from reading firmware, we can following the following method:

- Decap the chip ot expose the silicon
  - This subject is dealt with in depth in Chap 15
- Put a mask over the silicon to prevent blanking the flash

with the UV

- Carefully expose the relevant areas we want to reset
- Place the chip under UV light
- Enjoy reverse engineering your firmware!

So, eventually companies have grown wise to techniques such as this, and have begun to put some controls in place. One such example is found on the PIC 18F1320 - the security bits that control the read/write access to the firmware are covered by small metal squares. These are designed to prevent anyone resetting them and extracting the firmware. See figure 18.2 for a die photo and diagram about how this is supposed to work.

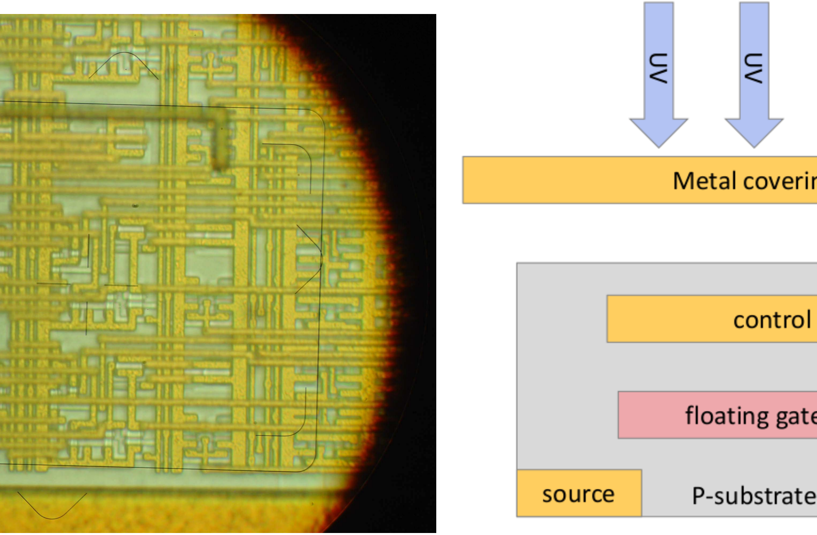

UV
UV
Metal coverin
control
floating gate
source
P-substrate

Figure 18.2:

Resetting via A Tilt

These controls, however, overlooked one key thing - light can be reflected by metal. It seems very obvious when you say it, but this is a very subtle attack - we proceed as the steps outlined above, however we now tilt the IC by 45° in order to cause the light to reflect off the bottom of the IC, then up to the metal covering, and reflect off this into the substrate in order to reset the FGMOS. Figure 18.3 has an illustration about how this is envisaged to work. Photographs of the attack can be found on Bunnie's blog - the the link above.

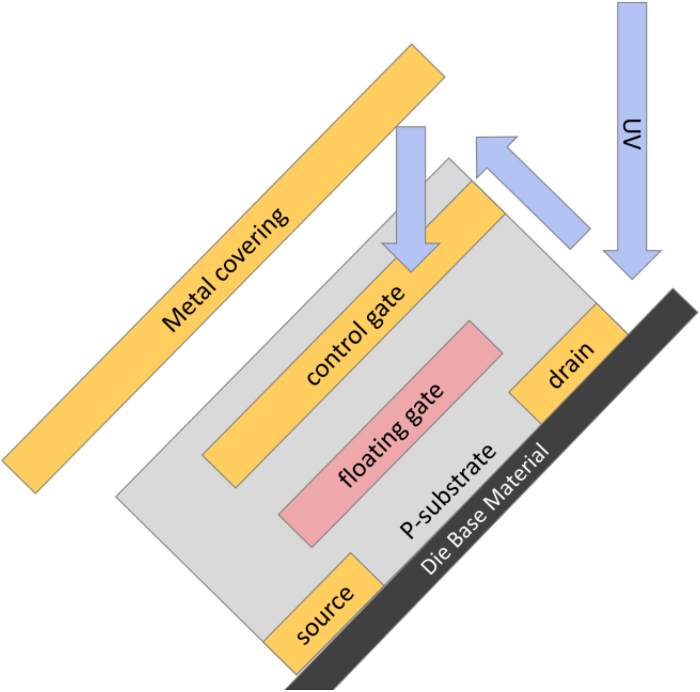

UV
Metal covering
control gate
floating gate
drain
P-substrate
Die Base Material
source

Figure 18.3:

This is a very elegant and simple solution to this particular problem of bypassing these controls - and although it requires some other work (such as decapping) in order to be effective, it is a very elucidating example of the kinds of approaches that have been employed to bypass CRP and other hardware controls.

18.2.2 LDR Register Readout on NRF51822

Background to the Attack

This attack comes from the Include Secuirty blog - `Source: http://blog.includesecurity.com/2015/11/NordicSemi-ARM-SoC-Firmware-dumping-technique.html`

ARM-based microcontrollers are all the rage. After 8051, ARM is the most in-use RISC architecture in embedded systems. In any given computing unit (a laptop, mobile phone, desktop PC, etc.) there are several embedded cores to any major CISC-based CPU.

Being a RISC architecture, there are some things that are done very differently - one of these things is how data is loaded. In x86, you can reference memory locations in opcodes without any issue. But on ARM, if you want to store or load a string you cannot simply access memory - you have to use two specialist opcodes that will store and fetch data for you. These are `STR` and `LDR` - the first one stores data, and the second one loads it.

`LDR` is a very straightforward opcode - here's the rough structure for those less familiar with ARM:

```
LDR r4, [r3, \#4]
```

Here, the value in `r3` is increased by 4, and then the contents of memory at the address corresponding to that value is loaded into `r4`.

This is a ubiquitous opcode, so being able to leverage it to bypass something like CRP is a real boon. Especially given the popularity of the target in this case, the NRF51822.[5]

Attack Method

To exploit this attack, we look to the following method:

1. Access debug on the target NRF51822
2. Locate a `LDR` opcode, and record its location
3. Work out which register it is reading from
4. Use the debug control to set the `PC` register and input register (`r3` in the above example)
5. Step the processor and retrieve the data from the output register (`r4` in the above example)
6. Repeat until the whole firmware is acquired

Assuming we have found the relevant opcode location and parameters, we can script this attack over OpenOCD's telnet server as follows in this python script:

```python
import sys
import telnetlib
import re

HOST = "localhost"
```

---

[5] This IC is used in devices like the BBC Micro:bit, and devices that need low power BLE, suchas fitbit clones.

```
PORT = "4444"
timeout = 500

tn = telnetlib.Telnet(HOST, PORT)
f = open("firmware.txt","w")

tn.read_until("> ")
tn.write("reset halt\n")
for i in range(0,0x200):
  # replace this '3' with the location
  # of the LDR opcode in the firmware
  tn.write("reg pc {0}\n".format(3))
  tn.write("reg r3 {0}\n".format(i))
  tn.write("step\n")
  tn.write("reg r4\n")
  num,_,lines = tn.expect([r'r4.*\w'], timeout)
  f.write(lines)
  #print(lines)
f.close()
```

Clearly this script will only get the first 512 bytes of firmware, but can be used to acquire whole images from microcontrollers such as the NRF51822.

Note - This attack works on ICs in the NRF51xxx series, but is known to not work on the NRF52xxx series ICs, so it may well be that this issue has been mitigated on newer chips.

18.2.3 'Heart of Darkness' Attack

This attack was originally due to Milosh Meriac `https://www.openpcd.org/images/HID-iCLASS-security.pdf` against the HID iCLASS.

This particular attack concerns the PIC18F452 IC - the CRP controls are quite strict. You can bulk-erase the bootloader or all of the storage blocks - but no further granularity of control. However, this is enough for us to proceed.

First we create a bootloader that prints out the entire firmware:

```
for(int i=0; i<0x8000; i++){
    printf("%x: %x", i, &i);
}
```

We now compile this and proceed as follows:

1. Bulk erase the bootloader, leaving the firmware intact

2. Load our own custom bootloader

    - Note - this will now dump the entire firmware through a UART connection, so we have the bulk of the firmware

3. On a second device, bulk erase the bootloader, and write the dumper code to the end of the firmware

    - Bulk erase replaces the firmware with NOPs, so the bootloader will send execution to the firmware entry-point, but this will be a NOP sled that will be ridden all the way down to the dumper code
    - We have now acquired the bootloader firmware from the device

Figure 18.4 has two diagrams that illustrate the two main phases of this attack.

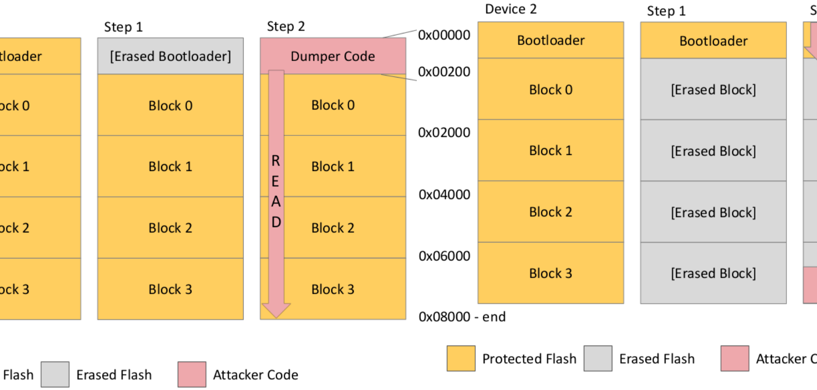

Step 1
Step 2
tloader
ock 0
ock 1
ock 2
ock 3
[Erased Bootloader]
Block 0
Block 1
Block 2
Block 3
Dumper Code
READ
Flash
Erased Flash
Attacker Code
Device 2
Step 1
0x00000
0x00200
0x02000
0x04000
0x06000
0x08000 - end
Bootloader
[Erased Block]
Protected Flash
Erased Flash
Attacker C

Figure 18.4: Phase 1 is on the left, phase 2 is on the right

This attack is a very nice example of how, with some careful management, a CRP can be bypassed even with a bulk erase being the only real tool we can use. The paper this attack first featured in is well worth a read!

18.2.4 Cold Boot Stepping

This attack was first put forward by Obermaier: `https://www.usenix.org/system/files/conference/woot17/woot17-paper-obermaier.pdf`

Background to the Attack

This attack instead chooses to focus on the memory of a microcontroller, as this can sometimes be left unaffected by CRP restrictions. As such, if a firmware does some sort of integrity check on itself, or some other process that ranges over the memory, this can possibly be leveraged to extact the firmware.

In the example presented, a CRC32 check is done on the firmware on the chip, and we abuse the level of access granted by a debug connection to control the flow of this function and pull the firmware, one double word at a time, out of memory.

This is the case on the STM32F051 - the memory is left unaffected from being read in full, even when CRP is fully enabled on the chip to prevent reading the firmware. As such, this is the example used in Obermeier's paper.

Attack Method

Here is an outline of the attack - firmware examples and code are all available in the paper and accompanying materials linked to above. Here is the general methodology of the attack:

1. Get debug control of the CPU

2. Identify when an integrity check is in progress

    - This is tricky, but fingerprinting the number of steps needed and the register behaviour for a CRC32, for example, and then comparing this to a target would be relatively straightforward.

3. Once this has been identified, we then setup a similar process to the LDR attack:

    (a) Let the check run for the first address `0x0001` and retrive the double word of firmware from that address from memory.

    (b) Reset the chip, and let it run to the second address, `0x0002`, and then retrieve the memory contents again

    (c) Repeat this incrementally for the entire firmware.

Figure 18.5 has a diagram demonstrating the rough mechanics of this attack. It is quite slow - Obermeier's example has the attacking STM32 chip emulating SWD in order to control the clocking very accrately, as well as the actual debugging. However, there is nothing that would explicitly prevent this from being effective over OpenOCD in some circumstances.

This attack highlights very well the kind of creative thinking that is very useful to aim to achieve - indeed, the kind of hack-y thought process, tenacity, and resourcefulness that is successful on limited systems such as those found in IoT/embedded devices hones many skills that are very transferrable within the tech industry.

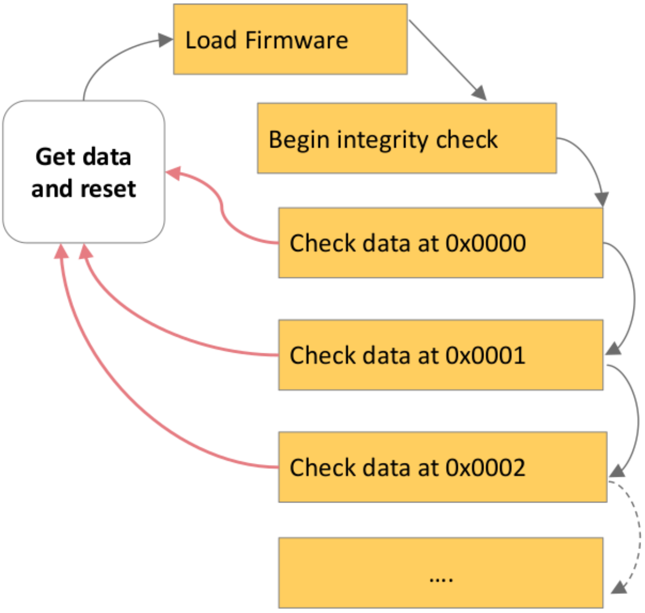

Load Firmware
Begin integrity check
Get data and reset
Check data at 0x0000
Check data at 0x0001
Check data at 0x0002
....

Figure 18.5:

Chapter 19

Cryptography on IoT/Embedded Systems

19.1

19.1.1

Appendix A

GDB Cheatsheet

Here is a GDB Cheatsheet of useful command/ways to do things in GDB:

Command	Description
`info <var>` or `i <var>`	print some information
`i b`	Get info about breakpoints
`i r`	dipslay all registers and their values
`i proc map`	display the process map information the code being analysed
`break label`	break on function label, e.g. `bre` `main`
`break *0x1000`	break on `0x1000`
`delete 2`	delete breakpoint 2
`set $r0=5`	set the value of register $r0 to 5

`stepi` or `si`	step forward one instr.
`nexti` or `ni`	step forward one instr., and ST OVER any calls
`stepi 5`	Step forward 5 instr.
`nexti 5`	step forward 5 instr., stepping over a calls
`continue` or `c` or `cont`	continue on with the process
`fg`	as in BASH, foregrounds the proce (push to background with `c&`)
`bt`	‘backtrace’ - show the call stack. **N** useful when analysing exceptions
`run arg1 arg2`	execute the debug target (from cli) w arg1 and arg2 as arguments.
`print system`	print the system symbol table da (stored as the `system` variable GDB) **NB** - very handy for ROP expl ing.
`p system`	Shorthand for `print system`
`p *(array-var)@20`	print 20 bytes of array-var
`x 0x1000`	short for ‘eXamine’, examine 0x1000 this case

`x/nfu 0xADDR`	Starting from 0xADDR, examine the 1. (N)umber of bytes, in a specifi 2. (F)ormat (any `x/d/u/o/t/a/c/f/s` fr C's print formatting, with ad tional `i` for instructions - defau to he(x)adecimal), 3. Specified (U)nit size (any (b)yte, (h)alfword, (w)ord, (g)iant/double words) e.g. `x/4dw $ps` is 'examine a dress in `$ps`, and print 4 words signed integer values' [Yeah, that on a mouthful...]
`x/10i $pc`	disassemble the next 10 instructio from the addr value in PC
`x/s 0x1000`	display the string at memory ad 0x1000
`x/10xb 0x1000`	display 10 bytes, as hex, starting addr 0x1000
`x/10xw $sp`	Show the 10 32bit words, in hex, sta ing at `$sp`
`set *(int*)0x6000 = 42`	Set an int (32bits) at addr 0x6000 to
`disp <expression>`	do ¡expression¿ each time a brea point is hit, e.g. `disp x/10i $pc`

`dissassemble (label)`	dissassemble the function with the bel specified, e.g. `disassemb` `main` will dissassemble main.
`disassemble 0x1000 0x2000`	disassemble from 0x1000 to 0x20 (replace with addresses in the p gram)
`set print pretty on`	set print formatting of C structures (use `off` to turn off)
`set logging file /path/to/file`	set logging to go to `/path/to/fi` from your session
`set arm force-mode arm/thumb/auto`	Set the arm forced instruction mo (arm/thumb/auto) - overrides the sy bol table
`show arm force-mode`	Show the current force instructi mode

Useful Commands

I also find these commands useful when debugging (on linux/posix like environments):

Command	Descr.
`gdb file`	Debug file in gdb (You never know, I mig forget...)
`gdb a.out 789`	attach GDB to `a.out` running as proce 789
`gdb a.out core`	open core dump for a.out in GDB
`cat /proc/789/maps`	get address space layout for 789
`echo 0 >/proc/sys/kernel/randomize_va_space`	turn off the pesky ASLR...
`execstack -s foo`	Disable XN for `foo`
`ulimit -c unlimited / ulimit -c 0`	turn core dumps on / turn core dumps off

Glossary

This will be a basic glossary of commonly looked-up terms.

BGA - Ball Grid Array. The IC package where blobs (balls) of solder underneath the IC are used to bond it to the PCB contacts.

CPE - Consumer Premises Equipment. Often, routers, but can also be IPTV boxes, DECT phones with base units, etc. Anything supplied by some service provider and left in a consumer's home.

PCB - a Printed Circuit Board. The thing with all the components on. Can be multilayered, or simply double-sided. Power and signals are communicated via traces on the PCB that link up components.

SMC - Surface Mount Components - the little tiny components that sit atop/underneath a PCB, and you daren't put down in case they get blown away by a flea's fart.

SOIC - Small Outline Integrated Circuit. A very common IC Package you will see on many consumer electronics.

Solder Mask - The part of a PCB that's coloured (green/black/red/whatever). This actively resists solder being put on it, driving it through surface tension to stick to the copper contacts and components instead.