556.2

Exploiting IoT Hardware Interfaces and Analyzing Firmware

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCEPTING THIS COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, USER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree, User may return the Courseware to SANS Institute for a full refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There’s an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

SEC556.2 | IoT Penetration Testing

SANS

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Welcome to SANS Security SEC556, *IoT Penetration Testing*. In this course, we'll take a deep look at the threats as well as the attack and defense opportunities surrounding wireless networks. Our primary focus in the course will be on evaluating IoT devices but we'll take a more holistic approach and examine much of the IoT ecosystem, including the hardware, networks, popular wireless protocols, and messaging services.

Let's keep this session interactive. If you have a question, please let the instructor know. Discussions about relevant topics are incredibly important in a class like this, as we have numerous attendees with various levels of skill coming into the class. Share your insights and ask questions. The instructor does reserve the right, however, to take a conversation offline during a break or outside of class in the interest of time and the applicability of the topic.

As the course author, we welcome any comments, questions, or suggestions pertaining to the course material:

Larry Pesce
larry.pesce.556@gmail..com
Twitter: @haxorthematrix

James Leyte-Vidal
jameslvsec556@gmail.com
Twitter: @jamesleytevidal

Table of Contents

Each course book includes a table of contents for each of the major topics and exercises that we'll cover.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. **Background and Importance**
2. **Opening the Device**
3. **Examining and Identifying Components**
   Exercise: Obtaining and Analyzing Specification Sheets
4. **Discovering and Identifying Ports**
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

Course Roadmap

In addition to the table of contents, each course book includes a course roadmap to show our progress as we work through each topic and its exercises. The current topic will be underlined. Previous topics will be shown in a light gray typeface to indicate completion.

Why Do We Care about Hardware?

IoT ecosystems are full of various components

- Software, mobile, web…
- …hardware

Hardware devices in the field for

- Human interaction
- Sensing
- Display
- System control

Hardware influencing behavior of humans and systems

Why Do We Care about Hardware?
In the IoT ecosystem we find a wide array of components from front and backend end software, mobile applications across multiple platforms, web applications on the devices itself and at the manufacturer or cloud services.

The IoT ecosystem is depended on hardware components for interaction with networks and humans. We also see them as interactive displays or interacting with other network and system components. The hardware platforms serve as essential parts of the IoT ecosystem affecting both the behavior of the users, or the networks or systems to which they are interfacing with.

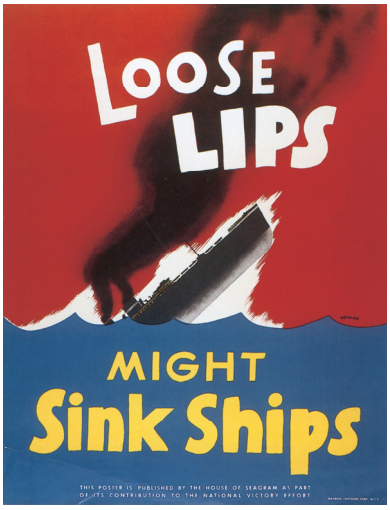
In the Hands of Many…

IoT hardware sold to consumers, enterprise, manufacturing…

Hundreds of thousands, if not millions of devices

Placed in the hands of millions of humans

Once sold and in the hands of a consumer, all control of the device by the manufacturer is lost



In the Hands of Many…
As soon as the devices are sold and placed in the hands of consumer any secrets that they hold are likely compromised, as the devices are now in an untrusted environment.

Millions of these devices placed in the wild, means that some percentage will be investigated for functionality, across all industries.

Physical Access to Pivot

In the hands of humans, they have physical access.

Physical access is game over for security implementation.

Starting point for determining

- Physical tampering.
- Software abuse.
- Repurposing for other applications.

With access, attackers can gain enough knowledge to turn the devices into network pivots.

Physical Access to Pivot

One of the tenets of computer security is that physical access is key, or a "game over" situation. When an individual has physical access to a computing device, the only thing that is a restrictive to eventually compromising the device is time. Given enough time, every device will fall victim and secrets will be recovered, and the device will be compromised.

This compromise often is the result of physical tampering (or "hardware hacking"), or local software abuse looking for command injection or OS manipulation. Once compromised the hardware can be repurposed for other uses, some with minor changes, others with a massive overhaul and complete switch of overall functionality.

With successful attacks and device changes, attackers can leverage the compromised IoT devices as network pivots into other sensitive networks, exploring them for options for physical control of other devices, or access to other sensitive network resources.

Linksys Case Study

Linksys WRT54G in hands of consumer

Discovered command injection via web interface

Lead to command line access, discovery of OSS components

Turned into a huge industry for device hacking, network implants, attack platforms and huge attack surface

Linksys Case Study
One of the most famous stories of hardware hacking, repurposing, and the advent of the hardware/IoT implant comes from the Linksys WRT54G line of routers, later acquired by Cisco.

The initial devices were found to be vulnerable to command injection on the built-in web interface, ultimately allowing access to the underlying operating system. Initially, much of the complaint was over the use of OSS components, but this turned onto a renaissance of device repurposing, modification.

This ultimately inspired an industry of device hacking, modification and turing devices into dedicated network pivots, forever expanding the attack surface for IoT devices.

An amazing history of the Linksys WRT54G story can be found at https://tedium.co/2021/01/13/linksys-wrt54g-router-history/.

Summary

Hardware is essential to the IoT ecosystem
Placing the device in the market grants physical access to attackers
Huge attack surface, industry changes

Summary

In this module we learned about the importance of hardware devices to the IoT ecosystem, and how once they are out of the manufacturers control, they can reveal secrets and be used as an attack platform for physical control, or other network access. These forays created industries for creating attack devices, as well as creating a huge attack surface for adversaries.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. **Opening the Device**
3. **Examining and Identifying Components**
   Exercise: Obtaining and Analyzing Specification Sheets
4. **Discovering and Identifying Ports**
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

Opening the Device

IoT/IIoT features hardware; lots and lots of hardware.

Each device is enclosed in pretty packaging.

The inside electronics are where the magic happens.

- Operating systems
- Radios
- Storage

Opening your device WILL void your warranty, but it is so much fun!

Opening the Device

IoT is reliant on hardware devices deployed across the globe in the millions. Each one of these devices has an attempt ads some "packaging" to be aesthetically pleasing, or functional in an attempt to protect the inner workings and circuit boards.

Those inner electronics components are what make the IoT/IIoT devices function and can reveal a lot about the functionality, purpose, secrets about function and the overall security implementation of the device itself and how it interacts with backend cloud services. If we can inspect the magic inner workings, we can gain access to the device Operating System, various storage components, radio chipsets and others depending on device function.

In order to access them, we need to open the device. In almost all cases, opening the device will void our warranty. With expensive devices, this is potentially terrifying! However, many distributed IoT devices are relatively inexpensive.

Components are where the ecosystem we find a wide array of components from front and backend end software, mobile applications across multiple platforms, web applications on the devices itself and at the manufacturer or cloud services.

The IoT ecosystem is depended on hardware components for interaction with networks and humans. We also see them as interactive displays or interacting with other network and system components. The hardware platforms serve as essential parts of the IoT ecosystem affecting both the behavior of the users, or the networks or systems to which they are interfacing with.

Tools of the Trade

Strength!
Blade, knife, X-acto
Spudger, prying tools
Screwdrivers
For the advanced deconstruction

- Drill and bits
- Rotary tool with cutoff wheel
- Files/sandpaper
- Chemicals

Tools of the Trade
Opening the device often requires the use of some simple tools, the simplest being our hands and some brute strength. There are many opportunities where the cases are simple enough, and friction fit with just a small manipulation.

Other simple tools include some sharp blade for cutting labels, and even for removal of some epoxy, glue, or conformal coating. Prying tools, and "spudgers" are helpful for separating case components, and it is good to use the right tools for the right job, as to not damage the device, or yourself! We also will find that many cases are fastened together with screws of various sizes and types, so having a robust set of screwdrivers is important.

For advanced disassembly, additional tools may be needed for more difficult to open devices with cutting tools, such as a drill with set of drill bits, a rotary tool with cutoff wheel, and even some abrasion with files and sandpaper. The most advanced disassembly can require the use of dangerous chemicals to dissolve potting or epoxy but is a method that is not for the faint of heart (not to mention dangerous!).

The spudgers featured on this slide can be found at https://www.amazon.com/gp/product/B06XNPRL42/ and https://www.amazon.com/gp/product/B01DGNCNR0/.

Screws

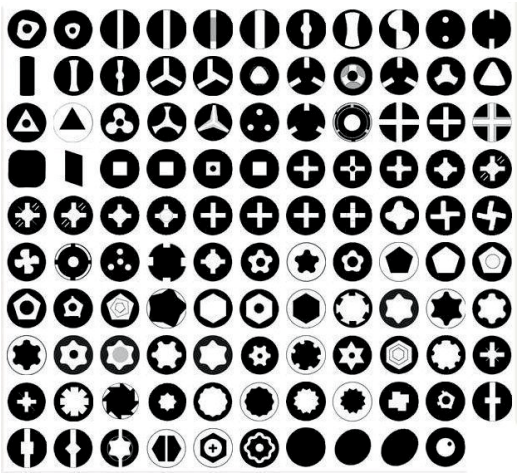
Most commonly found are Phillips screws of all sizes

Also common, Torx and security Torx, multiple sizes

Plenty of other security screws

- Tri-wing
- Snake eyes
- Triangle
- ...many others

Keep honest users honest



Screws
Screws are very common for fastening our cases together, the most common variety being the standard Phillips head screws of various sizes (typically #1, #2 and smaller). Torx and security Torx (Torx with a post in the center) are also quite common.

There are easily 50 different security screws available, but most are note seen in all but the extremely high end, or security conscious manufacturers. Security bits are well documented, and in most cases readily available in kits or individual for reasonable costs. As a result of the availability, the security screws only provide a temporary barriers to keep unmotivated adversaries out. As a final resort, drilling out screws is always an. (albeit destructive) option.

A simple set of security bits can be found at https://www.amazon.com/gp/product/B00BTKFFAU/.

Destruction!

Sometimes no obvious opening methods

Common in

- ICS/IIoT
- Automotive
- Weatherproof devices

Sealants

Potting/conformal coating

Bonded/welded metal and plastics

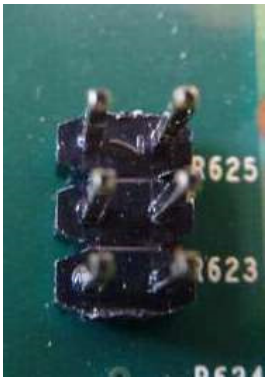
Destruction!
Sometimes the most fun part of doing a teardown to gain access to the magic insides is some destructive access techniques. Sometimes there is no other way to gain access, than to fold, spindle or mutilate the casing to get inside.

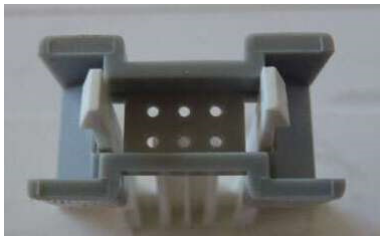
Destructive disassembly is most often found where devices are installed in harsh environments, or where there is an interest in keeping casual observers out. This can be done by permanently bonding all parts of the case together through metal or plastic "welding", application of sealants for wet weather, or encasing in epoxy or non-conductive conformal for weather and dustproofing.

In the example photographs on this slide, the author has used a rotary tool with cutoff wheel to remove two welded halves of a proprietary RFID tag meant to be exposed to the elements as part of heavy equipment or fleet management.

Tamper Detection

Uncommon, limited use
Self-destruct jumpers
Light sensors
Magnetic/reed switches
Accelerometers





Tamper Detection
In rare cases, the manufacturers will include some tamper detection and prevention techniques, but they are typically few and far between/ Depending on the level of sophistication, and device application we may observe the use of light sensors to trigger when the case is opened, or even magnetic/reed and tilt switches, to detect opening or manipulation for a device to be installed in a fixed location such as a power meter.

In extreme cases we can encounter "self-destruct" jumpers, as shown on the example images on this slide. The jumpers are placed in a random pattern into a holder affixed to the side of the case, so that when the case is opened the jumpers are unplugged and the original order cannot be determined. Without the jumpers present in the correct order, the device fails to function. Some additional reading for a case study on self-destruct jumpers found in "the wild" can be found at https://www.edn.com/teardown-inside-a-3g-microcell/.

Summary

Opening our device can be tricky
Simple tools, some specialty
Be careful of tamper detection!

Summary

In this module we learned some of the various tools and methods for opening out devices to begin our internal exploration. Most tools are readily available, and have common substitutions, but more robustly secured devices may require the use of specialized tools, also affordable in cost. Each model of device opening will be different with varying levels of complexity, including limited cases where tamper detection is found.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. **Examining and Identifying Components**
   Exercise: Obtaining and Analyzing Specification Sheets
4. **Discovering and Identifying Ports**
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

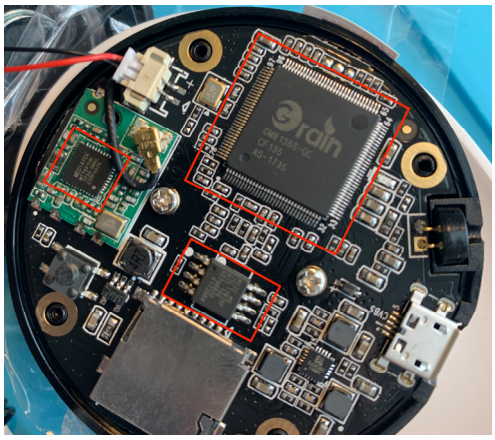
Examining and Identifying Components

Starting point for determining
- Device functionality
- Operating voltage
- Components for appropriate interaction

After opening device, inventory, and record IC identifiers
- Pictures, lots, and lots of pictures!

What is their purpose?

What is their connectivity to the other devices?



Discovering and Identifying Components
In order to understand how we can interact with a device and determine some if its's operating capabilities, we need to examine the various components on the board, record and perform an inventory if the chipsets.

This inspection process will assist us in understanding device functionality and capabilities, including those cases where the device is equipped with components that are not yet implemented in the operating system or device feature set; as an example, first generation Nest thermostats included a ZigBee radio on the board, but at time of release to the marketplace it did not support any use of ZigBee, where later software updates made use of the feature.

Some other helpful things we can learn from the board and chip/Integrated circuit (IC) inspection of the overall device functionality, feature sets, and how the components communicate together. We can also learn about the operating voltage, which can be helpful in tuning some of our tools and performing interaction with the device: powering a 3.3V device with 5v can let the magic smoke out!

Getting up Close

Many ICs with labeling

Labeling often subdued coloring, viewable in certain lighting condition

Small, small text

Use magnification!

- Modern cell phone camera with post capture zoom
- DSLR with macro lens
- USB connected microscope
- Head mounted magnification

Getting up Close

After disassembly we are going to find that there are several components that bear identification. We are going to find that the dark colored ICs are often embossed or printed with some very muted colors and tiny text. Depending on the overall size, these markings can be difficult to read especially with poor lighting conditions. It is best to examine the components in a well-lit area, with even some additional focused lighting, paired with some magnification.

Magnification can come in many flavors, including the use of a cellphone camera and zooming in after the fact, or even a good DSLR camera with a macro lens. an inexpensive USB connected microscope ($20-$120) is also an option, and with a real time video feed, it us also useful for soldering and other manipulation. Finally, if we don't need to document with photographs, a head mounted magnifier is also a great solution. Even if you look slightly nerdy while using it.

The Head mounted magnification featured on this slide can be found at https://www.amazon.com/gp/product/B07WJ36XBZ/ and a reasonably priced USB connected microscope can be found at https://www.amazon.com/gp/product/B08K7FGY9Q/.

Inventory

Capture photo of whole board, print

Document all numbers

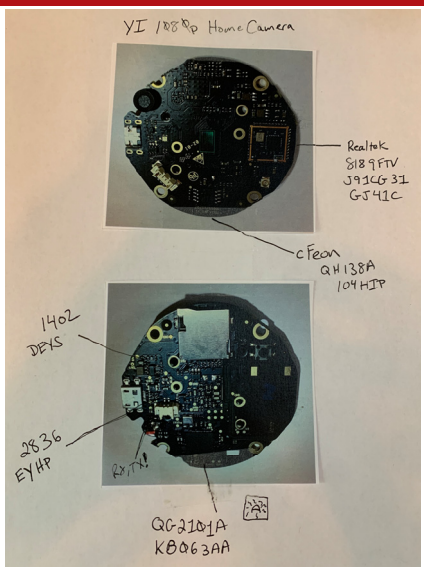
- Not all are important!
- Do we know which ones are not?

Document logos

Label the whole board and document what is where!

Capture and document both sides of all boards

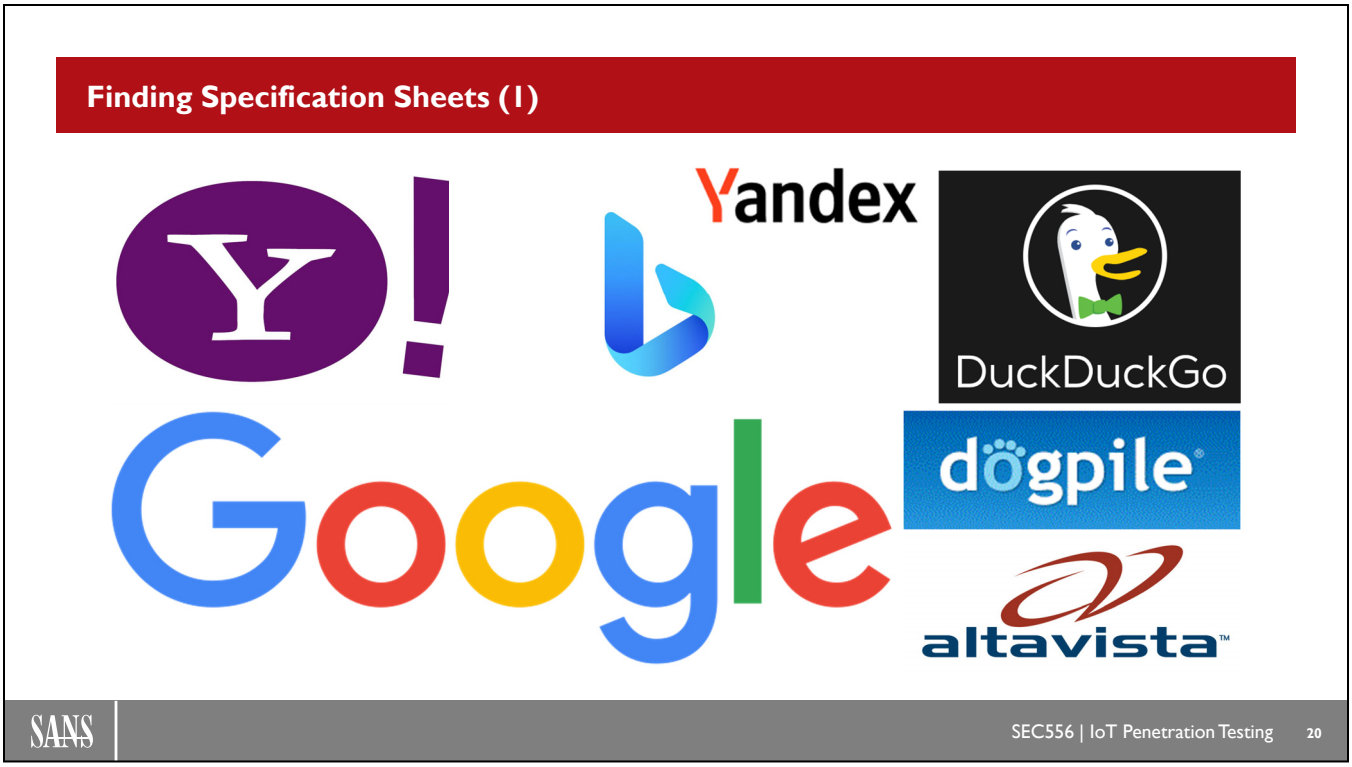
Now to find the spec sheets for each component…



Inventory

One approach to thorough documentation and inventory is to photograph both sides of the device printed circuit board (PCB), print them out and label. This provides a quick, readable visual guide to help with investigation.

With our documentation, it is important to document as much of the text on each chip as possible as we don't know which will be important for our lookup and unique model number during out datasheet lookups. Document the logos; we don't need to be master artists, we just need enough to compare to some images.



Finding Specification Sheets (1)
These notes intentionally left blank. No, seriously, just Google it.

Finding Specification Sheets (2)

Really, Google is your friend!

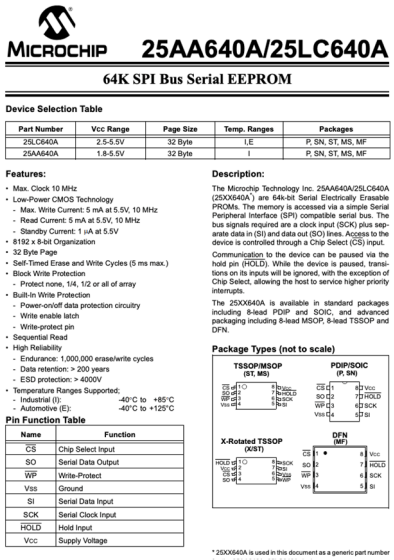
Search for partial part numbers, manufacturer (if you have it)

Compare logos on image searches

Many sites will have them for free

- Straight from the manufacturer
- Distributors both domestic and foreign

What happens when we can't find one…?



MICROCHIP 25AA640A/25LC640A

64K SPI Bus Serial EEPROM

Device Selection Table

Part Number	Vcc Range	Page Size	Temp. Ranges	Packages
25LC640A	2.5-5.5V	32 Byte	I,E	P, SN, ST, MS, MF
25AA640A	1.8-5.5V	32 Byte	I	P, SN, ST, MS, MF

Features:

- Max. Clock 10 MHz
- Low-Power CMOS Technology
  - Max. Write Current: 5 mA at 5.5V, 10 MHz
  - Read Current: 5 mA at 5.5V, 10 MHz
  - Standby Current: 1 µA at 5.5V
- 8192 x 8-bit Organization
- 32 Byte Page
- Self-Timed Erase and Write Cycles (5 ms max.)
- Block Write Protection
  - Protect none, 1/4, 1/2 or all of array
- Built-In Write Protection
  - Power-on/off data protection circuitry
  - Write enable latch
  - Write-protect pin
- Sequential Read
- High Reliability
  - Endurance: 1,000,000 erase/write cycles
  - Data retention: > 200 years
  - ESD protection: > 4000V
- Temperature Ranges Supported;
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C

Description:

The Microchip Technology Inc. 25AA640A/25LC640A (25XX640A*) are 64k-bit Serial Electrically Erasable PROMs. The memory is accessed via a simple Serial Peripheral Interface (SPI) compatible serial bus. The bus signals required are a clock input (SCK) plus separate data in (SI) and data out (SO) lines. Access to the device is controlled through a Chip Select (CS) input.

Communication to the device can be paused via the hold pin (HOLD). While the device is paused, transitions on its inputs will be ignored, with the exception of Chip Select, allowing the host to service higher priority interrupts.

The 25XX640A is available in standard packages including 8-lead PDIP and SOIC, and advanced packaging including 8-lead MSOP, 8-lead TSSOP and DFN.

Pin Function Table

Name	Function
$\overline{CS}$	Chip Select Input
SO	Serial Data Output
$\overline{WP}$	Write-Protect
Vss	Ground
SI	Serial Data Input
SCK	Serial Clock Input
$\overline{HOLD}$	Hold Input
Vcc	Supply Voltage

Package Types (not to scale)

TSSOP/MSOP (ST, MS) · PDIP/SOIC (P, SN) · X-Rotated TSSOP (X/ST) · DFN (MF)

* 25XX640A is used in this document as a generic part number

Finding Specification Sheets (2)

The best way to find our specification sheets is to use a search engine of our choosing. Google happens to be the author's search engine of choice and usually a result can be found within the first page of results.

Begin your search with the first portion of the IC identifier and the manufacturer if it is available. Text searches will often find PDFs from one of the large US electronics distributors, as well as some of the large non-U.S. ones. Image searches can also give us the opportunity to compare logos for manufacturers.

We will also find that even the manufacturers of the individual chips will also provide them: This author finds that there is often much more documentation direct from the manufacturer, and the spec sheet we are looking for is much harder to find.

What happens when we can't find a specification sheet? Typically, one of a few things have happened: We mis-identified the chip read with similar or hard to read characters, meaning we should search for character variants. Also, it is entirely possible that the chip was a specific design for the device (often in ICS equipment), and the specification sheets are not available to the general public. In these cases, if we can identify the manufacturer, we can reach out to them to ask for it; or work with the implementer (especially If we are under contract and an NDA) to obtain a copy.

A Brief Case Study

We observe an alleged flash chip

Labeled as Winbond 25Q128JVSQ 1504

Spec sheet, right in the title gives us some great information!

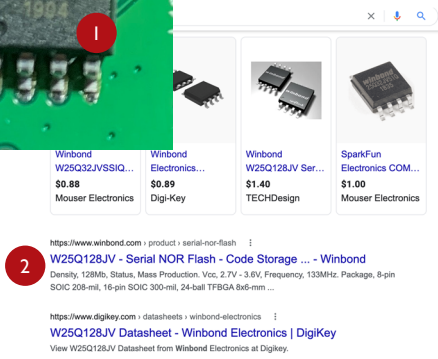
Additional read will give us commands for read/write, pinout

W25Q128JV

winbond

spiflash

3V 128M-BIT
SERIAL FLASH MEMORY WITH
DUAL/QUAD SPI
3



A Brief Case Study

In this example, disassembling a device we find several ICs of interest. One of our chips looks to be labeled as a Winbond 25Q128JVSQ 1504. A search at Google for "25Q128JVSQ", with some pretty immediate results! The second link at DigiKey, one of the large US based electronics distributors, has a direct link to the PDF of the specification sheet (https://www.digikey.com/en/datasheets/winbond-electronics/winbond-electronicsw25q128jv20spi20revc2011162016).

Loading up the PDF we can get a bunch of information right on the title page! Looking at the title we can see that it operates at 3V (likely 3.3V) is a Serial Flash Memory" module with SPI. We can now interact with the memory chip at 3.3V, with the SPI protocol.

Further reading will reveal pinout (which should be standard), any needs for read/write locking including trigger of pins or software commands as well as erase commands that can allow us to read protected portions of memory. In just a few minutes, we've had a huge success.

Summary

Chips/ICs all have specific function, not obvious
External markings helpful for discovery
Google (and others) are your friend!
Reading specification sheets detail functionality, protocols, pinout

- This will help guide our attack path

Summary

In this module we learned the value of obtaining specification sheets to determine chip functionality in order to determine additional attack path. Google or other search engines become our friends for locating the specification sheets, after performing an inventory of the chip labelling.

Exercise: Obtaining and Analyzing Specification Sheets

Using your lab book, complete the exercise, *Obtaining and Analyzing Specification Sheets*.

This page intentionally left blank.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. Examining and Identifying Components
   Exercise: Obtaining and Analyzing Specification Sheets
4. **Discovering and Identifying Ports**
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

Discovering and Identifying Ports

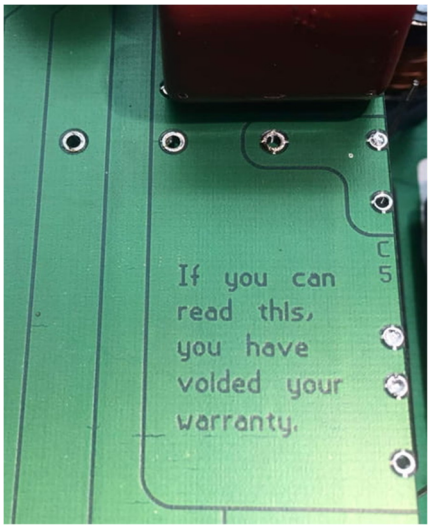
Many devices have pads available for interaction

Typically for manufacturer troubleshooting, factory programming, device access

Not all just "ports", some are access paths between components

- Possible interaction and sniffing data in transit

How do we find these access paths?



Discovering and Identifying Ports

In order to start interacting with the device, we need to start finding places on the PCBs where we can connect after opening the device. We'll be looking for connection points, or pads, which we can to interesting points of the chipsets identified earlier.

Typically, the connection points that we are looking for are broken out into some nicely formatted groups. These groups are most commonly used for the manufacturers for troubleshooting, debug, and initial programming of the device. We can use these same connection points in the same way (and better) than the manufacturers do!

Sometimes it isn't as easy as just finding a collection of well labeled pads. Sometimes we're following access paths looking for transitions in the circuit (vias) or points where they connect to other ICs. Tracing these paths can even give us the opportunity to sniff inter-chip interactions, recovering configuration information, or even sensitive data.

Visual Analysis

We can get lucky with easily identifiable, labeled circuit boards!

- Observable grouping of connection points on boards
- Observable grouping with silkscreen labeling

Some connection points labeled, but scattered

Typically, JTAG, Serial/UART, limited for others

Dear EEs: Stop labeling pads. Make us work for it!

Even better, no connector pads at all: build more complex test/programming rigs!

Visual Analysis

We will often get lucky with groupings of connection points right on the board, sometimes conveniently labeled with their function right on the labelling silkscreen! It is great when they are conveniently grouped together in "standard" formats, but they can also be spread out across the PCB. It still is easy when they are labeled, but they may not be!

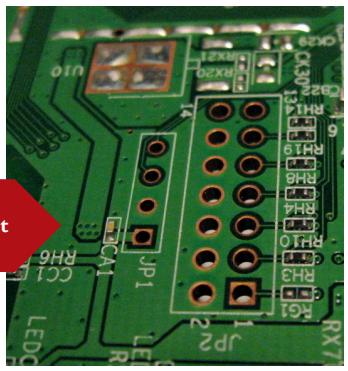
Most often we find connection pads (nicely grouped or otherwise) for JTAG and Serial/UART, as they are most commonly used for the manufacturer for programming the device before it leaves the factory, or for various levels of debugging. These test connections are used with a specialized test fixture, where the populated PCB can be placed in, powered on, and interacted with without the need for solder and wire every time, reducing the time to interact.

Our advice to make it more difficult to interact with the devices, is to stop leaving grouped and/or labeled test pads, intended for use with the programming test fixtures. Remove them and make us sped hours hunting for them! The manufacturers can build new, more complex test fixtures to utilize non-traditional test points along the path of transmission or direct to the IC mounting pads.

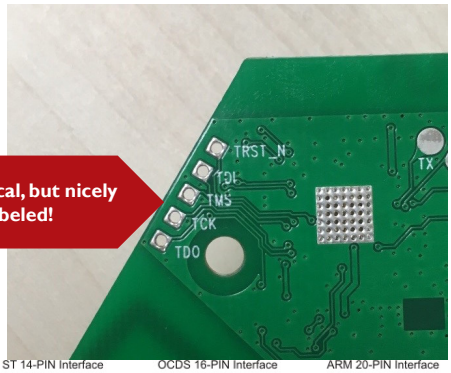
Visual Analysis: JTAG

Many JTAG interfaces have recognizable sizes and layout.

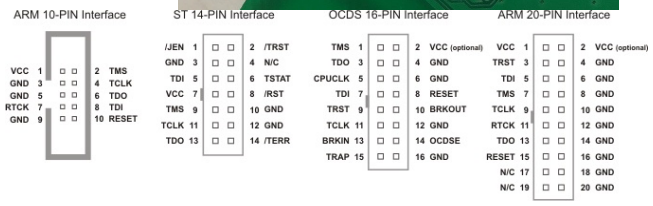
Several common layout formats



Common 2x7 layout



Not typical, but nicely labeled!



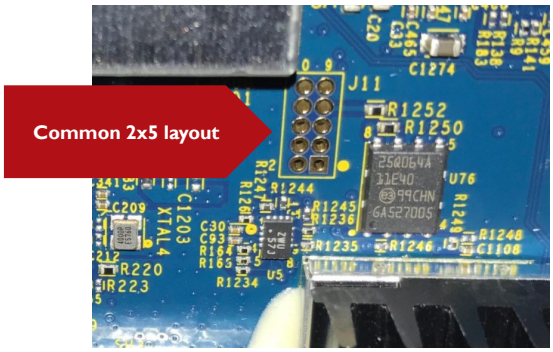
Visual Analysis: JTAG

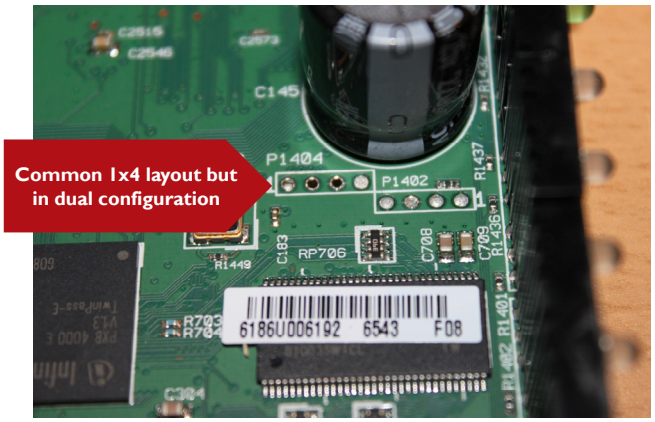
In the left-hand example of this device PCB, we see that there is a well-defined grouping of pads available matching a common, standard 2x7, 14-pin JTAG layout, with the square pad indicating pin 1. There are several common layouts for JTAG, depending on the application, chipsets used, and available space on the board.

On the right-hand side we can see JTAG broken out for us, and easily identifiable by the specific labels provided, even if they are not in a standard layout.

Visual Analysis: UART/Serial

Many UART have recognizable layout with common formats.





Visual Analysis: UART/Serial

In these examples, we have some commonly observed Serial/UART connectivity, again with pin 1 with a square pad or labeled in the silkscreen. Serial/UART doesn't typically have a standard format, but limited pins for interactivity is often a giveaway. On the common 2x5 layout on the left-hand example, often indicates two Serial/UART ports available; of course, the right-hand example may also feature two serial ports, from the observation of two sets of 1x4 connectors.

Visual Analysis: SPI Inspection

Typically, not exposed to a header
Sometimes we get lucky…

This example has SPI mapped to the header along with other functionality. Isolating SPI in this example requires following paths/traces on the board.

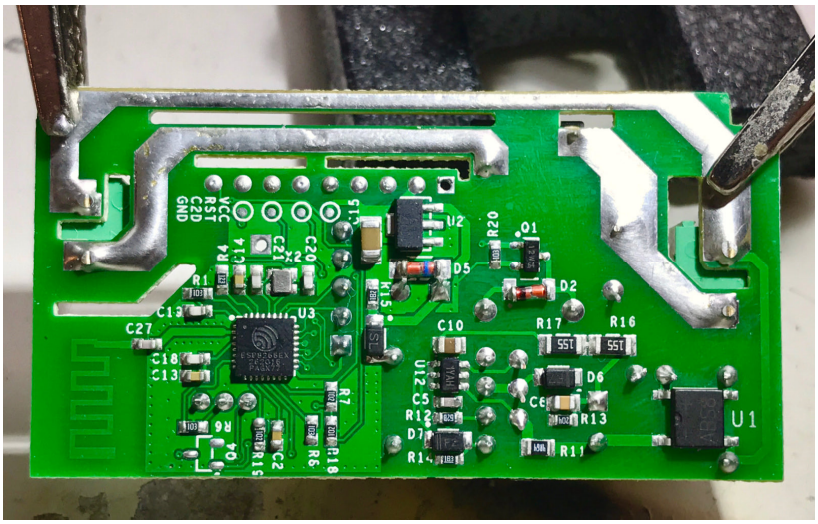
Visual Analysis: SPI Inspection
In this case, we have SPI broken out into a large header, which doesn't match the typical JTAG variants. In this example, we know that SPI exists at this header, because the work was done for us; some unidentified individual figures out IC functionality and followed the electrical paths (traces) to discover that they were populated at this header.

It is unusual to find SPI broken out to a header, much less labeled.

Visual Analysis: I2C

I2C also not typically broken out to headers

More tracing paths, deciphering chip pinouts, spec sheet analysis



Visual Analysis: I2C

In this case, we have I2C broken out into a small, but labeled header, which doesn't match the typical Serial/UART variants. Under normal circumstances, we'd be left to determining IC functionality and following the electrical paths to discover that they were populated at this header.

It is unusual to find I2C broken out to a header, much less labeled. The labelling has helped us determine the functionality.

Spec Sheet Analysis

If no access ports are observed, spec sheet analysis is our only hope.

Identify component functionality, pinout.

Observe paths on boards.

- Multi-layer boards make this more difficult
- Vias for board transition
- Continuity testing for tracing paths

Helpful for identifying JTAG, Serial/UART, SPI, I2C, …

- Direct access and sniffing as a result

Spec Sheet Analysis

It is quite common that devices that we want to interact with may not have any headers at all, with "randomly" places test pads. This leaves us to do some heavy lifting through identifying each IC on the board, their functionality, and the pinout in use of each IC. Once the functionality and pinouts have been determined, we are left to tracing the paths on the board for electrical continuity to determine use, connectivity and points to interact.

This operation is made more difficult with the advent of multi-layer (more than two) that hide the paths as part of the sandwiched layers, as well as the layer transitions (known as vias).

When we are successful, we set ourselves up for direct action to the device components, as well as sniffing of the various protocols in transit.

Summary

Common layout formats lead to port discovery
Standard layouts can have several forms for the same functionality
Obscured inclusion requires deep analysis

Summary

In this module we observed that in many cases the ports that we want to interact with are nicely laid out, and are easily recognizable, making out job of interfacing easy; on the converse, when they are not nicely laid out or labeled, figuring them out can be a time-consuming task.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. Examining and Identifying Components
   Exercise: Obtaining and Analyzing Specification Sheets
4. Discovering and Identifying Ports
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

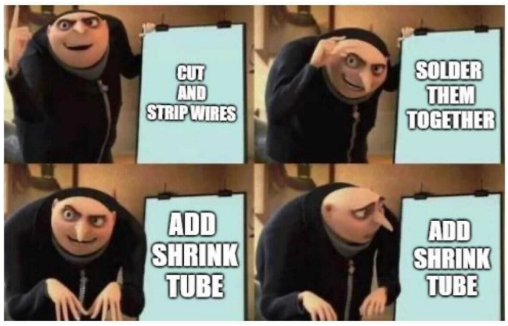
This page intentionally left blank.

A Soldering Primer

In many cases, our access to ports is not already enabled with header pins

- Empty through hole connectors
- Exposed test points
- Directly to surface mount or though hole IC pins

Soldering, or bonding pins and wires to these places grants us access



Soldering can be dangerous! It uses high heat, molten metal, and potentially dangerous supplies and fumes. Use appropriate safety gear, safe supplies, and fume extraction.

A Soldering Primer

We will often find, as observed in our port identification section, there were no header pins soldered into the headers. While this makes it easy for the manufacturer to access with the custom test fixtures through use of spring loaded, pointed pogo pins, it is not so easy for us to connect to them. If we solder some header pins to the headers, we have multiple reliable options for connectivity, from, test leads of various types, to Dupont wires.

We may need to access these test points that take several forms:

- Empty (or filled) through hole connectors in exposed headers
- Other exposed test points, which resemble a flat test point
- Directly connecting to IC pins, either those that are through hole (like a header), or surface mount pads (like flat test pads)

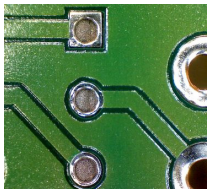
By soldering wires or header pins to these connection points gives us the ability to interact with the components and protocols.

Soldering is potentially dangerous stuff! It has several way sin which it can hurt us, so be careful and use appropriate safety precautions. Learn from one author's experience, getting burned and accidentally dropping a hot soldering iron in their lap while wearing pants made of synthetic materials.

Soldering Types

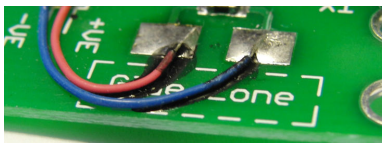
Through hole

- An actual hole through the circuit board!
- Wire, pin, or IC pin placed through hole
- Solder used to bind from the reverse



Surface mount

- Component and board mated surface to surface
- Solder used to bind on the obverse



We'll focus our efforts on soldering on exploitation as opposed to circuit construction. The overall techniques are the same!

Soldering Types

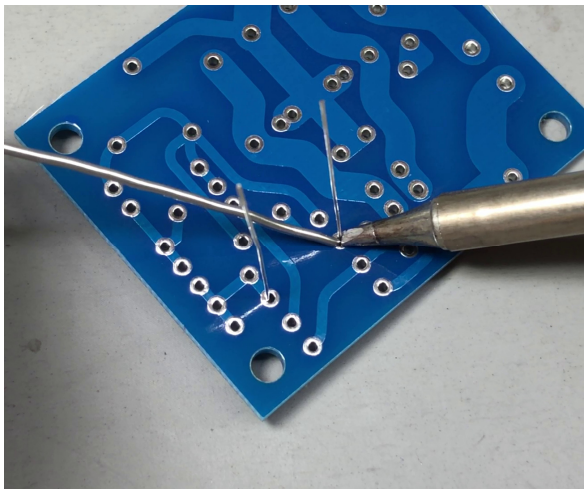
There are two variants of soldering that we can engage in: through hole or surface mount.

With through hole there is an actual hole through the PCB in which a wire or component pin is placed through and soldered on the reverse side of the board. Surface mount is a flat pad where the components is mated surface to surface, often on a much smaller scale. In this case, the solder is used to bon on the facing, or obverse side.

Soldering Technique: Through Hole

Heat the pad, pin/wire, and solder at the same time.

- Flux or flux core solder helps draw melted solder in
- No need to overdo it!



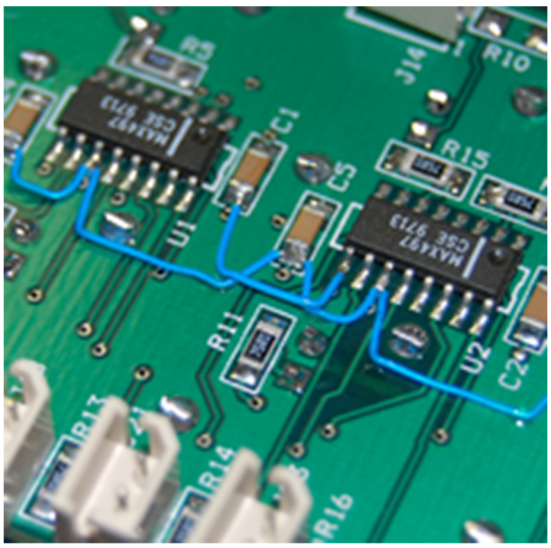
Soldering Technique: Through Hole
With through hole soldering, we are applying the tip of the soldering iron to the pin on the obverse of the board as well as the connective pad, while feeding solder to the top of the iron, effectively heating all 3 at the same time, in the shape of a triangle.

Application of flux, or flux core solder helps break the surface tension, and draw the melted solder to the components for proper bonding. We don't need to overdo it with tons of solder, as this is an electrical connection and not a mechanical one.

Soldering Technique: Surface Mount

Tin the attaching wire first!

Apply wire to top of pin/pad and heat from the top



Soldering Technique: Surface Mount

With surface mount we are creating more of a mechanical connection. As the solder is physically holding the component to the PCB. We use a different technique in that we apply solder to our wire (called tinning) and some solder to the pad or surface mount IC pin. We're not applying solder to the connection, but merely re-melting solder we've already added to the pad. This is most effective when heating the wire, we intend to bond from the top, passing the heat down to the pad.

Summary

Through hole soldering
Surface mount soldering
Soldering can be dangerous and requires patience

Summary

In this module we learned about some various soldering types for both through hoes and surface mount, both are relevant to our tasks for interacting with hardware.

Finally, we're dealing with hot, molten metal and dangerous fumes. Soldering can be dangerous!

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. Examining and Identifying Components
   Exercise: Obtaining and Analyzing Specification Sheets
4. Discovering and Identifying Ports
5. A Soldering Primer
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

JTAG

Joint Test Action Group (JTAG) names the group that standardized the interface

Typically used for debugging, reading, and writing firmware

- Recovery of boot options
- Direct control of components, processor
- Set watchpoints, direct memory modification
- A great way to recover firmware from a running device!

Typical 4 pin interaction 5th optional

- TCK, TMS, TDI, TDO, optional TRST

Finding the appropriate connection pads is challenging…

JTAG

JTAG is colloquially used as a term for the debug connectivity to a processor for debug purposes and flash memory read/write operations. JTAG refers to the group that standardized the interface, which is multi-purpose and multi-protocol.

As a penetration tester JTAG interfaces can give us a wealth of knowledge specifically for halting the processor and downloading a firmware image from the device for additional analysis, as well as writing modified firmware back to the device outside of normal user space upgrade options. In some more complex analysis JTAG can allow us to debug the running programs and kernel, directly control the processor and connected peripherals, debugging applications, and modifying registers in RAM, and even recovery of boot options.

Often, the hardest part of interacting with JTAG is finding the appropriate 4 pins of interaction with an optional reset line:

TCK: **T**est **C**loc**k** - The Clock sets the timing for communications.

TMS: **T**est **M**ode **S**elect - Voltage on TMS determines the instructions that JTAG performs.

TDI: **T**est **D**ata-**I**n - Data into the IC.

TDO: **T**est **D**ata-**O**ut - Data out from the IC.

TRST: **T**est **R**ese**t** (Optional) - This optional signal is used to reset JTAG to a known good state, effectively a ”JTAG reboot”.

JTAGulator

JTAGulator from Grand Idea Studios to the rescue!

Connect JTAGulator pins to unlabeled, possible JTAG ports

Interact with JTAGulator over USB serial

- Pick the pins in use on the JTAGulator
- Trigger fuzzing
- Observe responses to determine pinout for JTAG port

Not just for JTAG: ARM SWD and Serial/UART as well!

JTAGulator

An amazing tool from Joe Grand at Grand Idea studios in order to assist in JTAG connectivity discovery is the JTAGulator. It discovers JTAG pins by fuzzing/brute forcing signals to connect to the JTAGulator pins, in attempt to identify the responses. It is simple as connecting a bunch of unknown test points or headers and set the JTAGulator to do its job. However, the more test points that are connected, the more combinations that need to be tested, the longer the process will take.

To kick off the JTAGulator discovery process, after test points are connected, we interact with it over the USB/Serial port to access the interactive menu. In the menu we select JTAG discovery, and indicate the number of connected pins, start the discovery and wait.

Because the JTAGulator sends signals, and observes responses, it is useful for more than just JTAG discovery. It can also discovery ARM SWD (ARM's JTAG equivalent) as well as UART/Serial ports too. It won't perform these tests in tandem but require a separate operation to perform.

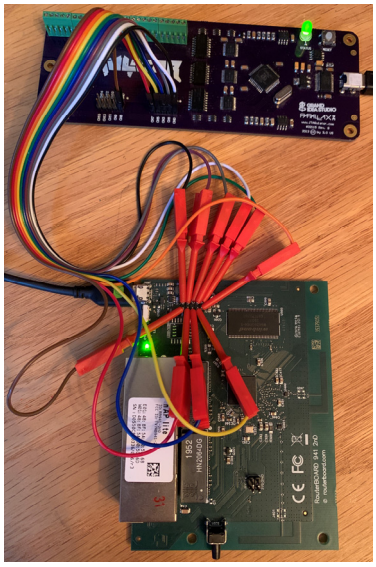
JTAGulator Example (1)

```
...<trimmed for brevity>
Welcome to JTAGulator. Press 'H' for available commands.
Warning: Use of this tool may affect target system behavior!

> H
Target Interfaces:
J    JTAG
U    UART
G    GPIO
S    SWD

General Commands:
V    Set target I/O voltage
I    Display version information
H    Display available commands

> I
JTAGulator FW 1.11
Designed by Joe Grand, Grand Idea Studio, Inc.
```



JTAGulator Example (1)
In this example, we have connected channels (pins) 0 through 8 to a header on a board and the JTAGulator ground pin to a known ground, and then powered on the victim device. Accessing the JTAGUlator over the USB ports as a serial device with 115200 baud, 8N1, we are presented with an ASCII art logo.

Entering H at the menu lists the available commands. Examining the running version of the JTAGulator software with the I command, we can see that we are running version 1.11, the latest available version as of this writing.

JTAGulator Example (2)

```
> V
Current target I/O voltage: Undefined
Enter new target I/O voltage (1.4 - 3.3, 0 for
off): 3.3
New target I/O voltage set: 3.3
Warning: Ensure VADJ is NOT connected to target!

> J
Enter starting channel [0]: 0
Enter ending channel [0]: 8
Possible permutations: 3024

Bring channels LOW before each permutation? [y/N]:
Press spacebar to begin (any other key to abort)
```

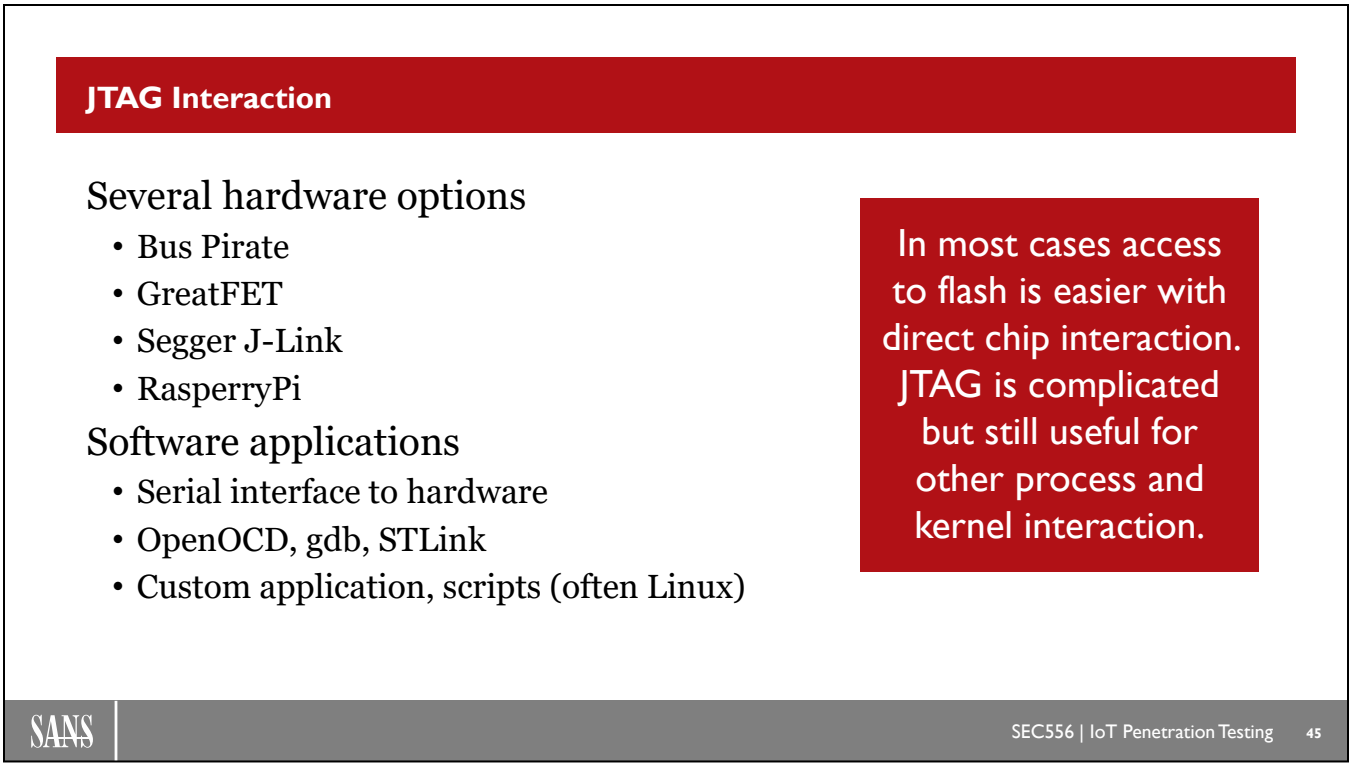
```
...<continued>
JTAGulating! Press any key to
abort...
----
TDI: 6
TDO: 7
TCK: 8
TMS: 3
Device ID #1: 0000 0000000000000000
00000000000 1 (0x00000001)
TRST#: 0
TRST#: 1
TRST#: 2
TRST#: 4
TRST#: 5

-
JTAG scan complete.
```


JTAGulator Example (2)

Continuing with the JTAGulator interface, we need to set the operating voltage of the device so that the JTAGulator can send and interpret responses appropriately. We set the voltage with the V command, and in this case, set the voltage to 3.3 volts based on the capabilities of the victim device.

Once the voltage has been set, we can use the J command to begin the JTAG fuzzing, by defining the channels (pins) on the JTAGulator that are connected to the victim device. Accepting the defaults for bringing channels low, we hit the spacebar to start the fuzzing. This example only took a few seconds (due to the limited number of pins tested), and the discovered pin output was returned, noting the JTAG functionality to the JTAGulator pins.



JTAG Interaction

There are many options for interacting with JTAG, some more robust, some easily obtainable. The Bus Pirate, GreatFET and RaspberryPi are all readily available and affordable, while interaction via the Bus Pirate and GreatFET are quite mature in their offering, the RaspberryPi is the newcomer to the game. The most mature is the Segger J-Link but you get what you pay for as it is often perceived as the most expensive.

Depending on the platform used we'll access them with either a separate serial interface, control with OpenOCD and gdb, or Segger STLink. We may even perform some automation with some custom scripting.

It is possible, and highly effective for us to dump contents of flash to recover firmware over JTAG, it is admittedly slow. It may be more effective to directly access the flash chip over SPI or I2C.

TCK: **T**est **C**lo**ck**
The drummer, or metronome that dictates the speed of the TAP controller. Voltage on this pin simply pulses up and down in a rhythmic, steady beat. On every "beat" of the clock, the TAP controller takes a single action. The actual clock speed is not specified in the JTAG standard. The TAP controller accepts its speed from the outside device controlling JTAG.

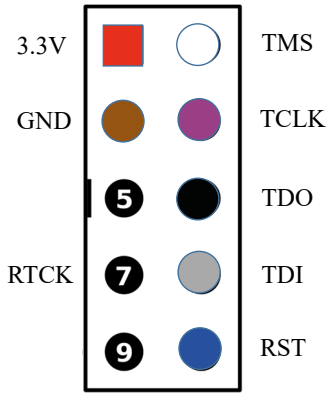
TMS: **T**est **M**ode **S**elect
Voltages on the Mode Select pin control what action JTAG takes. By manipulating the voltage on this pin, you tell JTAG what you want it to do.

TDI: **T**est **D**ata-**I**n
The pin that feeds data into the chip. The JTAG standard does not define protocols for communication over this pin. That is left up to the manufacturer. As far as JTAG is concerned, this pin is simply an ingress method for 1s and 0s to get into the chip. What the chip does with them is irrelevant to JTAG.

TDO: **T**est **D**ata-**O**ut
The pin for data coming out of the chip. Like the Data-In pin, communication protocols are not defined by JTAG.

TRST: **T**est **Rese**t (Optional)
This optional signal is used to reset JTAG to a known good state, we'll explain why this is optional in a few paragraphs.

Connecting to JTAG with Bus Pirate



RST is often optional, and RTCK is often unused.

This example shows the common ARM 10-pin connector. Researching datasheets on the device processor will determine which standard/format we should wire for.

Connecting to JTAG with Bus Pirate
In this course we've chosen to utilize the Bus Pirate to interact with JTAG because of its maturity, ease of use for recovering firmware and its overall availability.

The Bus Pirate is quite easy to connect to the identified JTAG ports using the color-coded Bus Pirate cable to the appropriate JTAG pins. While the cable colors may not match, we can use Dupont wires to perform the same connectivity.

In this case, we've shown the common ARM 10-pin JTAG layout, but our actual implementation may be different.

Flash Recovery/Dumping JTAG with Bus Pirate (1)

Requires Bus Pirate, OpenOCD and gdb

```
$ cat MyBuspirate.cfg
source [find interface/buspirate.cfg]
buspirate_vreg 0
buspirate_mode open-drain
buspirate_pullup 1
buspirate_port /dev/ttyUSB0

$ sudo openocd -f MyBuspirate.cfg
Warn: Adapter driver 'buspirate' did not declare which transports it allows; assuming
legacy JTAG-only
Info : only one transport option; autoselect 'jtag'
Info : Buspirate Interface ready!
<trimmed for brevity>

$ telnet localhost 4444
```


Flash Recovery/Dumping JTAG with Bus Pirate (1)
In order to dump flash via JTAG with a Bus Pirate, we need to use a combination of OpenOCD and gdb. First, we need to provide some initial configuration for OpenOCD defining where it can find and configure the Bus Pirate interface.

Using sudo to start OpenOCD (for access to the USB serial port), we direct it to use the previously defined configuration. Once OpenOCD has started, we begin interaction with it via telnet, on the default port of 4444 on our local system.

Flash Recovery/Dumping JTAG with Bus Pirate (2)

```
$ gdb extended-remote :3333

> halt; reset init
MIPS32 with MIPS16 support implemented
target halted in MIPS32 mode due to debug-request, pc: 0x800064e0
JTAG tap: ath79.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000,
ver: 0x0)

> halt; flash probe 0
target halted in MIPS32 mode due to debug-request, pc: 0x800064e0
Found flash device 'win w25q128fv/jv' (ID 0x001840ef)
flash 'ath79' found at 0xbf000000

> halt; flash info 0
#0 : ath79 at 0xbf000000, size 0x01000000, buswidth 0, chipwidth 0
<trimmed for brevity>

> halt; dump_image flashdump.bin 0x00000000 0xF90600
```


Flash Recovery/Dumping JTAG with Bus Pirate (2)
In this example, we have connected our Bus Pirate to the appropriate pins of the victim device as indicated previously. We then start the gdb debugger to connect to port 3333 on our local system (the colon by itself preceding the port number indicates localhost). Port 3333 is an additional interface opened by OpenOCD to provide a proxy to the JTAG bus for gdb, through the Bus Pirate physical interface.

Within the gdb interface, we then halt CPU operation and reset the device to a “known” state of a “clean” boot.

After restart, we begin discovering what flash devices are available, by first halting the CPU, probing for connected flash modules, and then obtaining the flash memory parameters and capabilities.

Once we have determined the memory layout, we can again halt the CPU and use the dump_image command within gdb, providing an output filename and a start and end flash memory address.

Serial

Serial communications, RS-232
- Typically, 5v

In IoT, mostly serial via UART
- Typically, 3.3V application of serial
- UART as hardware can do RS-232, RS-485, RS-422

Interaction with running OS as a terminal

Pre-boot process interaction/interruption
- U-boot, others

Even inter-component, on board communication

Serial

In IoT hardware we also find a liberal application of serial communications. We typically think of serial communications as 5-volt RS-232, most serial communications are 3 volt based in IoT, referred to as serial UART communications.

The implementation of UART opens up options for other serial-based protocols over RS-232, including the use of RS-485 and RS-422.

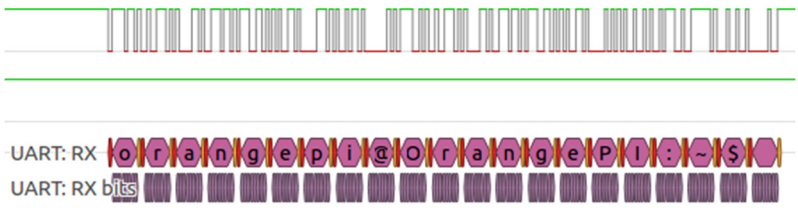
Interaction with serial UART becomes important for interacting with IoT hardware: often one of the exposed UART ports is enabled as a OS terminal. It we are able to interact with this terminal session at boot time, we should be able to interact with and interrupt the pre-boot process. These stages of the boot process, or boot loaders such as U-boot, prepare the system for loading the system kernel. The boot loader also feature methods for disk and memory debugging (including the ability to dump firmware) as well as the ability to engage failsafe mechanisms, such as uploading known good firmware/OS.

Additional opportunities exist for us for passive observation of serial signals, in order to sniff communications between components on the board. This on-board communication is often used by the OS to configures other components; observing that configuration can help us interact or compromise other interfaces.

Serial Architecture

Single data stream, typically un-synchronized

- No clock synchronization
- Synchronization information included in the stream
- Bits per character, start/stop bits: 8N1



Serial Architecture
Serial/RS-232/UART uses a single, un-synchronized data stream to transfer information, between two endpoints. There is no clock synchronization, as the timing is determined by the accuracy and timing of the data itself. This in stream data timing is determined by signal spacing, bits per character, defined and start and stop bits and parity. The most common variety is Identified as 8N1.

Serial Interaction

Serial/UART to USB adapter required
- Silicon Labs CP2104 based
- FTDI FT232RL based

Software terminal
- Windows: TeraTerm, PuTTY
- Mac: Serial (GUI), minicom, screen
- Linux: minicom, screen

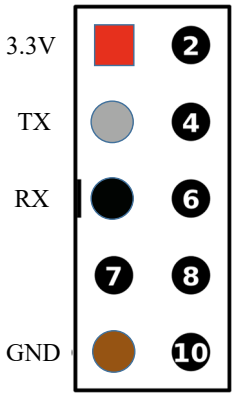
Need to discover on board and communication settings
- Direct interaction
- Sniffing with UART
- Sniffing with logic analyzer

Serial Interaction
We can start our UART interaction in two different ways: direct interaction with a terminal or sniffing existing communications. In both cases similar hardware tools can be used, such an UART to USB adapter. Common choices are the Silicon Labs CP2014 or FTDI FT232RL based devices, readily available for only a few dollars.

For software, there is an option for common our operating systems, either preinstalled with the base OS, easily installed with a package manager. Under windows TerraTerm and PuTTY, both available for free are popular choices, while Serial on macOS is $40 in the App store. On macOS and Linux, both minicom and screen are available for free.

After we find the port and pinout on the board, we can then directly interact with the serial port (hopefully finding a terminal already logged in), or sniffing the traffic already sent to another device. The sniffing with a UART to USB adapter is not the most robust way to sniff due to the way it can impact the electrical characteristics affecting the legitimate communication. For better sniffing a logic analyzer with OpenOCD may be a better choice.

Connecting to Serial with Bus Pirate



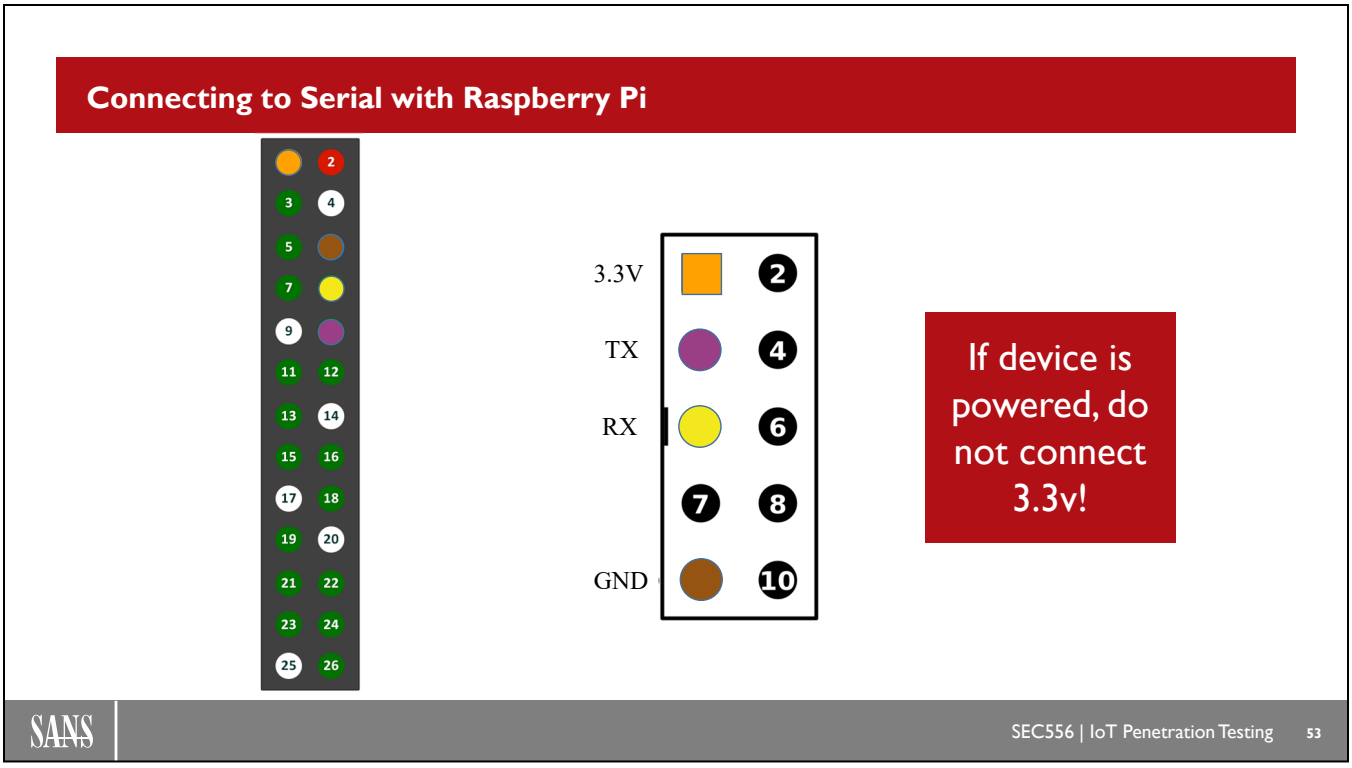
If device is powered, do not connect 3.3v!

Connecting to Serial with Bus Pirate

In this course we've chosen to utilize the Bus Pirate to interact with Serial because of its maturity, ease of use for recovering firmware and its overall availability, and its suitability for other tasks.

The Bus Pirate is quite easy to connect to the identified serial ports using the color-coded Bus Pirate cable to the appropriate serial pins. While the cable colors may not match, we can use Dupont wires to perform the same connectivity.

In this case, we've shown a common serial port header layout, but our actual implementation may be different.



Connecting to Serial with Raspberry Pi

Additionally, the serial port on board the Raspberry Pi can be used to interact with Serial port. We do need to be sure to disable the default terminal bound to the serial port before trying to access a device. Note, we need to edit */boot/cmdline.txt* on the Pi to remove "*console=serial0,115200*" or "*console=ttyAMA0,115200*" from the file, save and reboot. Alternatively, we can use raspi-config menus (from https://www.raspberrypi.org/documentation/configuration/uart.md:

Start raspi-config: sudo raspi-config.

Select option 3 - Interface Options.

Select option P6 - Serial Port.

At the prompt Would you like a login shell to be accessible over serial? answer 'No'

At the prompt Would you like the serial port hardware to be enabled? answer 'Yes'

Exit raspi-config and reboot the Pi for changes to take effect.

Note: Please do not perform these steps yet, as we be using the Raspberry Pi in our kit as a serial target and not as a receiver.

The Raspberry Pi is quite easy to connect to the identified serial ports using the color-coded Dupont wires to the appropriate serial pins. While the cable colors may not match, we can use Dupont wires to perform the same connectivity. Note that the diagram has the serial port ”purple” connected to pin 10 on the Pi, where “brown” is connected to pin 6 on the Pi.

In this case, we’ve shown a common serial port header layout, but our actual implementation may be different.

Serial Example (1)

Bus Pirate

```
...<menus trimmed for brevity>...
HiZ> m <- to change the mode
(1)> 3 <- for UART mode
(1)> 9 <- for 115200 bps
(1)> 1 <- for 8 bits of data, no parity control
(1)> 1 <- for 1 stop bit
(1)> 1 <- for Idle 1 receive polarity
(1)> 2 <- for Normal output type
UART>(0) <- to UART list macros
UART>(2) <- to start UART monitor
```


Serial Example (1)
In this example, we configure the Bus Pirate to use UART mode, and set N81 with 115200 data rate, giving us an interactive serial terminal for both sniffing and interaction.

Serial Example (2)

UART/RaspberryPi

```
$ screen /dev/ttyS0 115200 <- Select port and baud rate per system/UART adapter
```

Logic analyzer gui

Or command line

```
$ sigrok-cli -d fx2lafw --time 10000 --channels D0=RX --config samplerate=1m -P
uart:baudrate=115200:format=ascii
```


Serial Example (2)
Using the Raspberry Pi in similar fashion to the Bus Pirate is quite simple with the screen utility. We just need to define the default Pi serial port and a speed.

For the least impactful sniffing we'd want to use a logic analyzer. With the Sigrok GUI, after capturing data (1), we apply a decoder (2), select UART (3) and then observe the recovered data (4).

Alternatively, we can obtain text output from the command line, by applying a decoder as well. With the logic analyzer it is a read only operation with no interactivity.

SPI

Serial Peripheral Interface or SPI

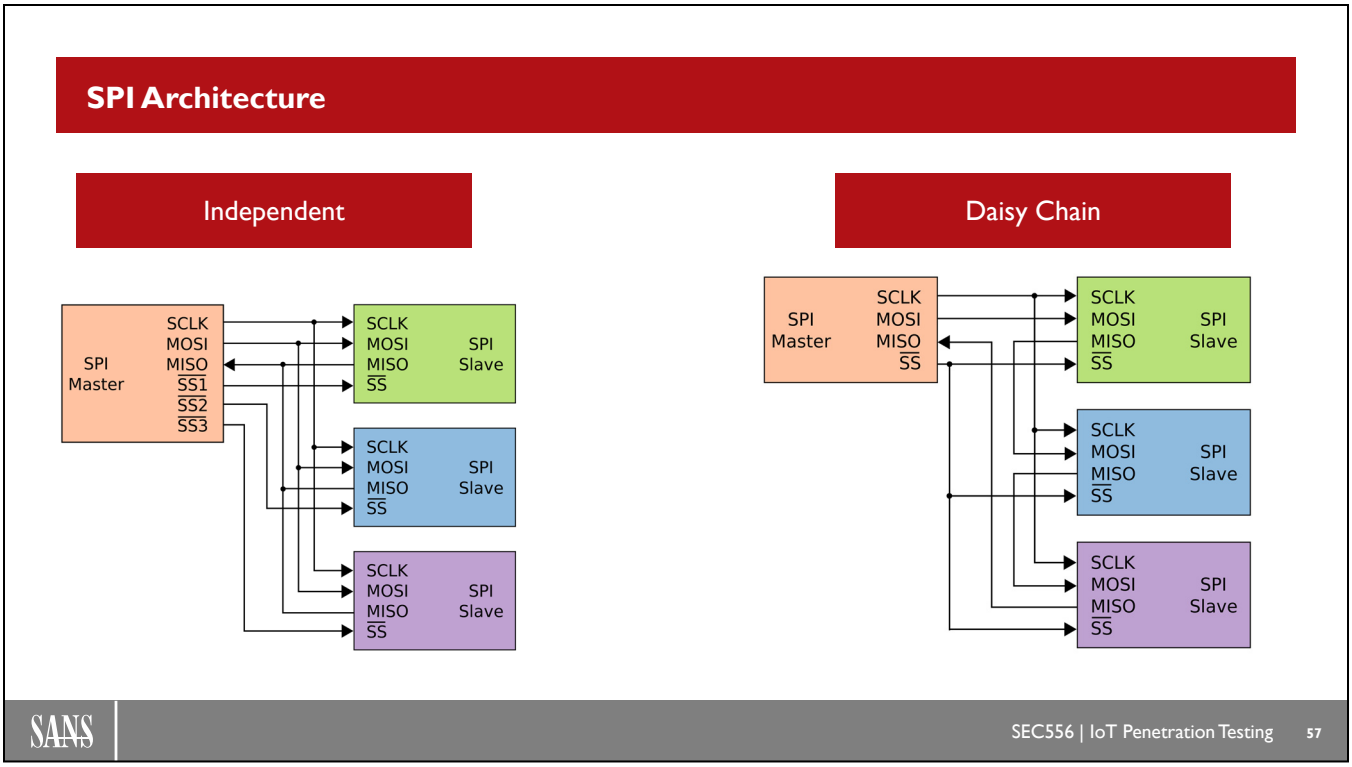
- Developed by Motorola in the 1980's
- Synchronous serial communication
- Full duplex

Speed determined by clock

SPI

We've discussed that serial is often popular for communicating on board between components. More often than serial, is an improved Serial Peripheral Interfaces, or SPI. Originally developed my Motorola in the 1980 in order to support full duplex serial communication in a synchronous manner, allowing for multiple devices to be enabled on a bus, the overall bus speed determined by built in clock timing.

We can sniff SPI and perform direct chip interaction. Interfacing with flash memory can be done over JTAG, but it is much faster directly with SPI.



SPI Architecture
SPI features 2 different architectures, independent and daisy chain.

In the independent architecture, each subordinate device has a dedicated SerialSelect (SS) line to the main device and a bus of the input output lines. If the main device needs to communicate with any particular serial endpoint, it activates the individual line by setting it high (enabled with voltage). In the daisy chain architecture, a single SS line is connected from the main to every serial endpoint, but the input and output lines cascade from one serial endpoint to the next.

When possible, and enough SS pins are available, the independent architecture is the most common, and fastest.

SPI Protocol

4-wire serial bus

- Master/Slave (Main/Secondary) architecture
- 2 architecture designs: Independent and Daisy chain

SCLK, MOSI, MISO, SS/CS/CE

- Serial Clock, Master(Main) Out Slave(Serial) In, Master(Main) In Slave(Serial) Out, Slave(Serial) Select/Chip Select/Chip Enable

SPI Protocol
Where traditional serial connections use 3 wires, in that they are point-to-point, the addition of the 4th wire supports the selection of which chip is enabled for communication on the SPI bus. As shown in our previous slide, the SPI bus can have two architectures; independent addressing and daisy chained.

The four lines for communication are SCLK, or the serial clock that sets the speed for the communication, MOSI, or main out serial in (TX, MISO or main in serial out (RX), and SS/CS/CE, or serial select, for selecting the destination from main.

SPI Interaction

Specialized hardware
- Bus Pirate
- RasperryPi
- GreatFET

Software
- Flashrom on RaspberryPi
- GoodFET client apps

Sniffing with Logic Analyzer

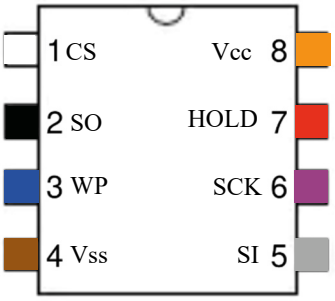
SPI Interaction

There are many options for interacting with SPI, some more robust, some easily obtainable. The Bus Pirate, GreatFET and RaspberryPi are all readily available and affordable, while interaction via the Bus Pirate and GreatFET are quite mature in their offering, the RaspberryPi is the newcomer to the game. Even though the Raspberry Pi interaction is less mature, the flashrom utility is quite robust and easy to use!

Depending on the platform used we'll access them with either a separate serial interface for the Bus Pirate or control with custom apps, for the GreatFET of GoodFET, and flashrom for the Raspberry Pi.

It is possible to monitor/sniff SPI communications, with OpenOCD in similar fashion to that with serial, by providing a different decoder.

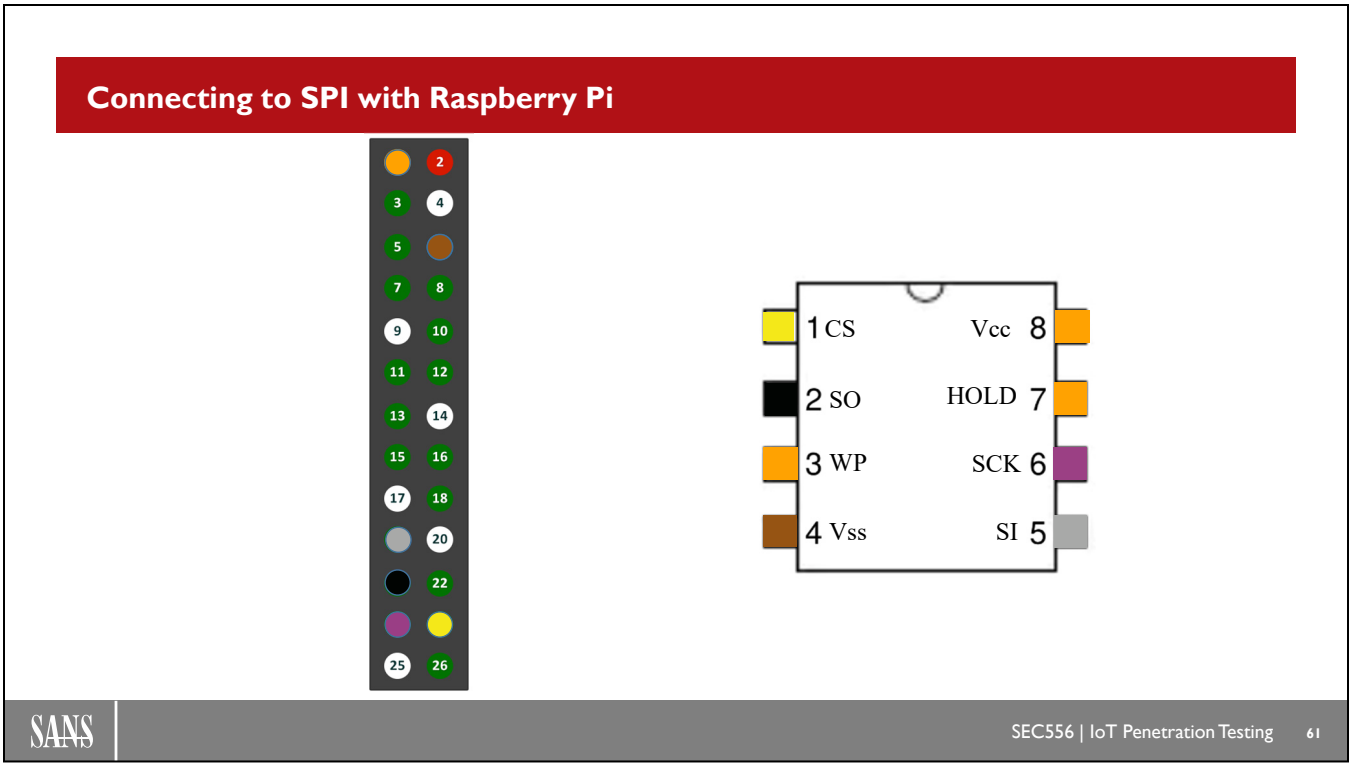
Connecting to SPI with Bus Pirate



Connecting to SPI with Bus Pirate
In this course we've chosen to utilize the Bus Pirate to interact with SPI because of its maturity, ease of use for recovering firmware and its overall availability, and its suitability for other tasks.

The Bus Pirate is quite easy to connect to the identified SPI ports using the color-coded Bus Pirate cable to the appropriate serial pins. While the cable colors may not match, we can use Dupont wires to perform the same connectivity.

In this case, we've shown a common SPI serial enabled chip pins, but our actual implementation may be different.



Connecting to SPI with Raspberry Pi
Additionally, the GPIO pins on the Raspberry Pi can be used to interact with SPI.

The Raspberry Pi is quite easy to connect to the identified serial ports using the color-coded Dupont wires to the appropriate SPI pins. While the cable colors may not match, we can use Dupont wires to perform the same connectivity. Note that the diagram has the SPI pin ”purple” connected to pin 23 on the Pi, where “brown” is connected to pin 6 on the Pi.

In this case, we’ve shown a common SPI chipset pinout, but our actual implementation might be different.

SPI Example

Bus Pirate

```
...<menus trimmed for brevity>...
HiZ> m <- to change the mode
(1)> 5 <- for SPI mode
(1)> 3 <- set speed to 250 kHz
(1)> 1 <- for clock polarity idle low (default)
(1)> 2 <- output clock edge active to idle (default)
(1)> 2 <- input sample phase end
(1)> 2 <- /CS (default)
(1)> 2 <- output type normal (H=3.3V, L=GND)
(1)> W <- to turn power supply on
(1)> [3 0 0 r:5] <- Read five bytes
(1)> A <- set AUX to HIGH, therefore /WP to HIGH, disabling write-protect
(1)> [6] <- Set write enable(WREN)
(1)> [2 0 0 1 2 3 4 5] <- Write five bytes
```


SPI Example

In this example we use the Bus Pirate interface over a serial port to read a few bytes and write a few bytes. In order to dump a full SPI chip to recover firmware, we'll need some scripting.

Flash Recovery/Dumping SPI

Bus Pirate can dump using SPI flash in an automated fashion with *flashrom*

Instead of individual read/write commands, flashrom can dump the entire contents at once

- Define driver, device, bus/read speed, and output file

RPi header	SPI flash
25	GND
24	/CS
23	SCK
21	DO
19	DI
17	VCC 3.3V (+ /HOLD, /WP)

```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M -r spidump.bin
```


Flash Recovery/Dumping SPI
Using the Raspberry Pi and flashrom, it is quite easy to dump the entirety of the SPI flash chip. We define the Linux SPI driver, define the device to use the appropriate GPIO pins, the speed and the output file.

In order to use SPI on the Raspberry Pi, we need to enable the functionality first using raspi-config.

sudo raspi-config

Select "Interfacing Options" :

Highlight the "SPI" option and activate "<Select>".

Select and activate "<Yes>" :

Highlight and activate "<Ok>" :

When prompted to reboot highlight and activate "<Yes>" :

The Raspberry Pi will reboot, and the interface will be enabled.

Alternatively, we can add some additional configuration options to the bottom of */boot/config.txt: dtparam=spi=onUse* sudo reboot.

Summary

Several common interface types, Serial, JTAG, SPI

Discovering the unknown with JTAGulator

Many interfacing options, Bus Pirate, Raspberry Pi, logic analyzer

Purpose built tools/software a robust option

Summary

In this module we reviewed the technology behind three common hardware interfacing options, for serial, JTAG and SPI. Of those connection types we learned about several hardware tools for connecting to them: While there are more expensive, purpose-built tools for interfacing with many of the identified connection methods, we also find that our arsenal can have more readily available, inexpensive options.

These common options include the multi-purpose Bus Pirate for Serial, JTAG, and SPI interaction, but we also encountered the Raspberry Pi for the same. Additionally, we observed where the use of a logic analyzer is helpful for observing data in transit, without an ability to interact.

Exercise: Sniffing Serial and SPI

Using your lab book, complete the exercise, *Sniffing Serial and SPI*.

This page intentionally left blank.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. Examining and Identifying Components
   Exercise: Obtaining and Analyzing Specification Sheets
4. Discovering and Identifying Ports
5. A Soldering Primer
6. Sniffing, Interaction, and Exploitation of Hardware Ports
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

Firmware Delivery

Firmware for IoT devices can come in one of several ways:

- Vendor website!
- Downloads over HTTP/HTTPS
- Auto-update functions from the device
- Download plus update 'push' over telnet/tftp/other insecure protocol

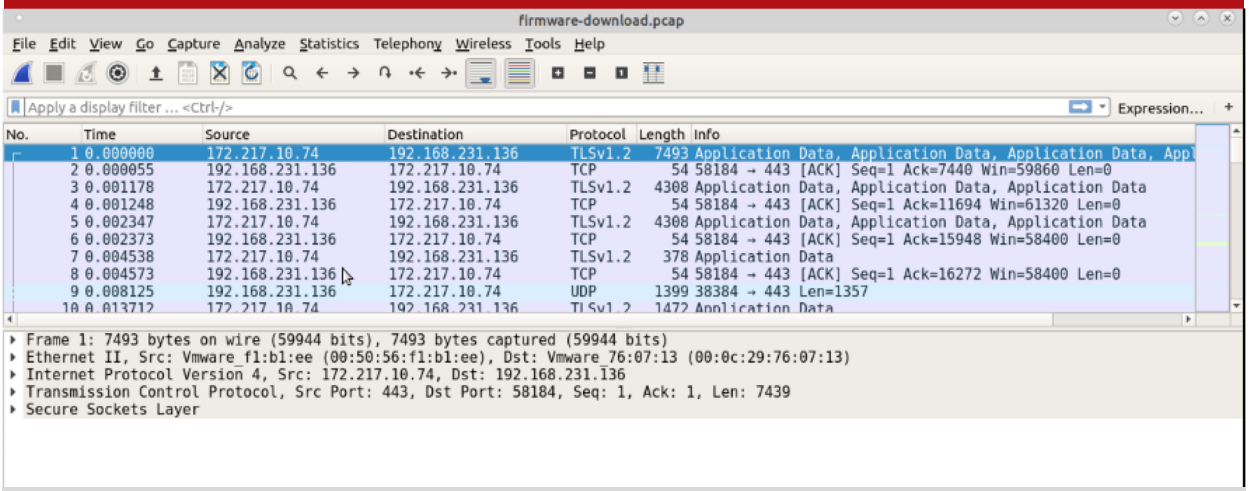
When not publicly available, firmware files are often delivered in an insecure fashion

Firmware Delivery

Accessing a fresh copy of firmware can simplify firmware analysis. In some cases, this is as simple as grabbing a copy of the firmware from GitHub, or the vendors website.

In other cases, we may need to grab the firmware via other means. Firmware may be sent over insecure protocols such as HTTP, FTP, TFTP, etc. which gives us an opportunity to retrieve a copy of this firmware over the air - if we set up a capture environment in advance!

Wireshark for Firmware Carving



We'll find this firmware in the next lab!

Wireshark for Firmware Carving

The amazingly capable protocol parsers in Wireshark allow for us to identify files being transmitted in a network capture and allow us to readily extract it.

Tshark for Firmware Carving

Like Wireshark, its cousin Tshark can help us find objects in a pcap as well

- Just add *–export-objects* for some automation

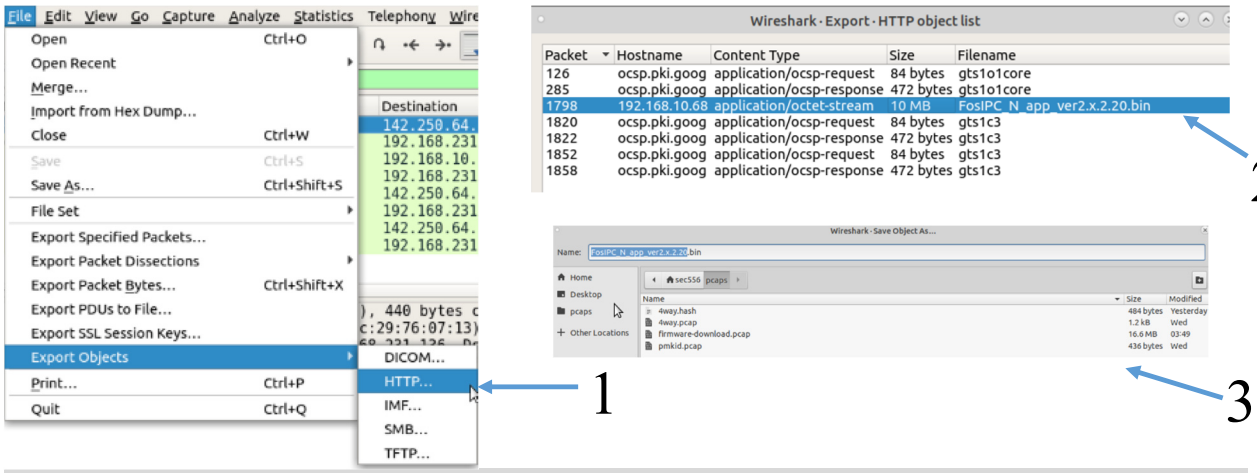
```
$ tshark -Q -r firmware-download.pcap -export-objects
```

We'll find this firmware in the next lab!

Tshark for Firmware Carving

Tshark, Wireshark's text-based cousin, can also help us extract objects from a packet capture for analysis as well.

Wireshark Process for Firmware Carving



We'll find this firmware in the next lab!

Wireshark Process for Firmware Carving

In Wireshark, once you validate the protocol used to distribute the firmware, you can use the Wireshark 'Export Objects' functionality to identify the objects sent over that protocol and select the ones of interest to download.

Summary

Firmware is not always available as a direct download

Network capture and extraction helpful

Summary

In this module we learned about other ways to capture firmware when it is not readily available from the manufacturer's website if we can observe an update. If we can observe the update and capture network traffic, it may be possible to extract the firmware from the network capture.

Exercise: Recovering Firmware from PCAP

Using your lab book, complete the exercise, *Recovering Firmware from PCAP*.

This page intentionally left blank.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. Background and Importance
2. Opening the Device
3. Examining and Identifying Components
   Exercise: Obtaining and Analyzing Specification Sheets
4. Discovering and Identifying Ports
5. A Soldering Primer
6. Sniffing, Interaction, and Exploitation of Hardware Ports
   Exercise: Sniffing Serial and SPI
7. Other Ways of Recovering Firmware
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

Firmware?

Every computing device needs:

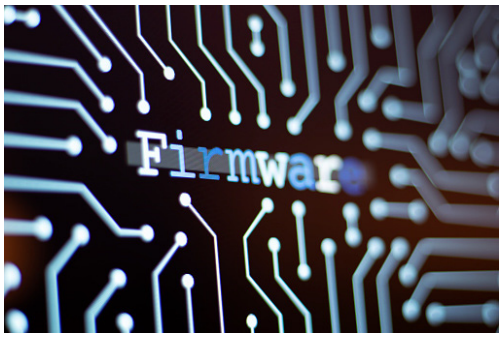
- Operating instructions (OS)
- Storage for applications
- System memory

Storage and memory provided by the hardware in RAM and Flash

Storage populated with OS, Filesystem by firmware

Bootloader provides base instruction set

- Initializes hardware, similar to CMOS, uEFI
- Points to location of kernel, filesystems



Firmware?
Every computing device requires a few components to operate; Operating instructions in the form of an operating system, some disk storage for the OS and applications, and system memory. In this case, IoT devices are no exception!

The disk storage in out IoT devices is often found in the form of flash memory and it mor permanent in nature. System memory, if found in the form of flash and is typically transient in nature. In IoT these are both found on the board as an IC. Specifically, the static, permanent storage is populated with a bootloader, Operating System, and a file system These are provided my one or more files, as a “firmware image”.

In this firmware image is the special bootloader, which initializes the hardware, in a similar fashion to BIOS or uEFI on modern desktop/laptop systems. This boorloader provides basic instruction to the CPU for hardware and to find the load points for the OS and kernel.

Firmware!

Typically, firmware is delivered as a file comprised of one or more disk structures.

Often structured filesystem with OS, storage

A single "blob" with disk partitions, OS, Applications

Contains everything needed for device operation

Obtaining a firmware image gives us hands on to a device.

- Minus the hardware, but the operating environment
- "Physical" access is "game over"

Firmware!
We find that manufacturers, regardless of IoT devices provide a single, monolithic firmware image comprised of multiple disk structures. These disk structures are often divided into separate partitions, for OS, read/write partitions and even bootloader.

The single file blob, with those filesystems often with different formats will contain the OS, applications for operations and everything it needs for operation, including configuration and stored secrets.

If we can obtain copies of the firmware, it is effectively giving us access to a snapshot of a system, minus the hardware. In the traditional IT world, this gives us physical access to the system, which is typically, "game over", granting access to all the secrets on the system. Therefore, obtaining access to firmware from a running device via JTAC, SPI or I2C is extremely important.

Firmware Analysis

With a full OS and applications, we can discover issues in implementation.

We don't even need to boot up the OS.

Extract filesystems, mount, and explore

- Potential opportunity to modify and reload.

Discover all manner of security failures.

- Web interface authentication issues, RCE
- Application and script failures
- Backdoor accounts
- Storage or private keys, certificates

Each device will have its own unique issues.

Firmware Analysis

If we obtain firmware, we get access to the running system where we can observe implementation and security issues. We can begin examining the OS without even booting up; booting, even in a virtual environment introduces a whole host of challenges from missing hardware to CPU architecture mismatches and more.

Instead, the single blob file can be broken down into its constituent parts. If we extract the file systems, we can mount them as a disk, examine the contents and even with the possibility of performing modification and reinsert the modification into the firmware and reload it back to the device.

With the filesystem analysis, we are afforded the opportunity to discover all sorts of implementation failures in the operating system, utilities, and custom application; from eeb app issues, with command execution and RCE, to application and custom script manipulation issues, and all manner of authentication issues.

Extraction and Verification

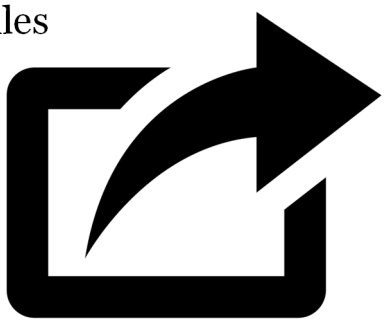
Some images are delivered as a compressed files

ZIP, LZMA, etc.

Manual extraction required

- Many do not indicate type of compression
- Manual analysis of magic numbers needed

After decompression, extraction



```
$ file firmware.bin
firmware.bin : u-boot legacy uImage, jz_fw, Linux/MIPS, Firmware Image (Not compressed),
11075584 bytes, Tue Sep 10 08:49:03 2019, Load Address: 0x00000000, Entry Point:
0x00000000, Header CRC: 0x59B3A1D8, Data CRC: 0xCD7CAB52
```


Extraction and Verification

Some firmware images that we acquire for analysis are compressed in order to make the file transfer to the device faster/smaller at time of flashing or update. Once uploaded to the device, the update process will extract the firmware.

We typically don't know that the image is compressed, and most end users don't care. It only becomes a hurdle in performing the analysis and extraction of the individual filesystems. The compression can be any one of the common types, including zip/gzip, bz2, lzma, tgz, etc.; however, we usually don't know the type by file extension. If we are extracting the filesystems, we are left to discover the compression time on our own. This discovery can be performed with the file magic numbers which indicate file type. The *file* command knows about whole host of magic numbers and can easily perform that analysis, giving us an indicator for tooling to use to perform the decompression.

Once decompressed, we can extract the filesystems and begin the hunt.

Zip Password Brute Force with Hashcat (1)

Hashcat with GPU acceleration is *much* faster than fcrackzip (if it works at all)

Requires diligent identification of ZIP application type

Application of hashcat's mask attack methods

Extract zip file hashes with zip2john

- Adding some command line text processing for proper hashcat formats

```
$ zip2john firmware.zip | cut -d ':' -f 2 > zip_hashes.txt
```


Zip Password Brute Force with Hashcat (1)

On limited occasions, especially with more secrets conscious manufacturers, the zip file could be protected with a password. This password would be known to the hardware device, in the form of the previously installed operating system. This provides a dependency (or chicken and egg) problem: the decompression password is stored in the firmware protected by that password.

These passwords are often not easily guessable, so some application of force is required. We can apply some brute force with hashcat, with either GPU or CPU based password guessing. Using the power of the hashcat mask based brute force attacks.

We do need to do a little work up front in order to guess some passwords! First, we need to extract some hashes from the zipfile using *zip2john* and pass it to *cut* to obtain appropriate output for hashcat. Unfortunately, (?) hashcat recognizes more than a dozen password protected zip file formats. We either need to know much more detailed information about the zip format or perform several operations to guess.

Zip Password Brute Force with Hashcat (2)

Application of Hashcats' mask attack methods

Marker	Character Sequence
?l	abcdefghijklmnopqrstuvwxyz
?u	ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d	0123456789
?s	«space»!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
?a	?l?u?d?s
?h	0123456789abcdef
?H	0123456789ABCDEF

ID	Zip Type
11600	7-Zip
17200	PKZIP (Compressed)
17220	PKZIP (Compressed Multi-File)
17225	PKZIP (Mixed Multi-File)
17230	PKZIP (Mixed Multi-File Checksum-Only)
17210	PKZIP (Uncompressed)
20500	PKZIP Master Key
20510	PKZIP Master Key (6 byte optimization)
23001	SecureZIP AES-128
23002	SecureZIP AES-192
23003	SecureZIP AES-256
13600	WinZip

```
$ hashcat -m 17200 -a 3 zip_hashes.txt "?a?a?a?a?a?a?a?a"
```


Zip Password Brute Force with Hashcat (2)

In order to perform mask attacks, with hashcat we need to make some educated guesses about length of the password, as well as the character set used. In this example we've elected to use a common PKZIP format (17200) with an 8-character, lower case password.

Manual Filesystem Extraction

Unix dd used for file carving

- *if* is the input file
- *of* is the output portion
- *bs* is the block size, 1 byte used for simplicity and accuracy
- *skip* is the start position
- *count* is the number of bytes to extract (end minus start)

```
$ dd if=firmware.bin of=jffs.bin bs=1 skip=808032656 count=17579298416239
```

Wouldn't it be nice if this was automated for us?

Manual Filesystem Extraction
If we crack the password, or start with an unprotected firmware, we need to start separating the parts; bootloader, and filesystems. We can use the *dd* command to perform the slicing of the firmware into its parts. We need to specify a few options:

- *if* is the input file
- *of* is the output portion
- *bs* is the block size, 1 byte used for simplicity and accuracy
- *skip* is the start position
- *count* is the number of bytes to extract (end minus start)

I n this case we have chosen to you 1-byte block size to make out math easy, however it significantly slows down the process. In addition, we need to know the start and end points and any built-in length indicators.

Binwalk

Manual extraction is tedious and subject to false negatives
Wouldn't some automation and better accuracy be welcome?
Binwalk from ReFirmLabs and Craig Heffner bring the real magic!

- Blob scanning for magic numbers
- Automatic file extraction, or manual with dd
- Automatic decompression (where possible)
- Filesystem extraction
- Entropy analysis
- Python API for extensibility, IDA Plugin

Binwalk

Our manual extraction of file systems if fraught with possibility for errors and omissions, aka false negatives. Not to mention it is lots of tedious, manual work. Any improvements in speed and accuracy are welcome in this process.

On too so drastically improve firmware filesystem extraction is *Binwalk* from ReFirmLabs and Craig Heffner (aka devttys0). Binwalk is amazingly capable for improving accuracy and speed with

Automatic scanning for magic numbers.

File extraction in an automated fashion, or manual with dd.

Automatic decompression.

Extraction and separation of filesystems into individual files.

Entropy analysis to examine for encrypted blobs.

Extensibility through a Python API and integration with IDA.

Signature Scanning with Binwalk

Binwalk can perform basic scanning of firmware
Discovers file headers and offsets for use with dd

```
$ binwalk camera-firmware.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC:
0x59B3A1D8, created: 2019-09-10 08:49:03, image size: 11075584 bytes, Data Address: 0x0,
Entry Point: 0x0, data CRC: 0xCD7CAB52, OS: Linux, CPU: MIPS, image type: Firmware Image,
compression type: none, image name: "jz_fw"
<output trimmed for brevity>
2097216       0x200040        Squashfs filesystem, little endian, version 4.0,
compression:xz, size: 3353204 bytes, 407 inodes, blocksize: 131072 bytes, created:
6225984       0x5F0040        JFFS2 filesystem, little endian
```


Signature Scanning with Binwalk

Partially automating this process is the use of *Binwalk* without any options to perform signature scanning. Signature scanning reveals the various components in the firmware blob. Binwalk will indicate the found "part" and the start and end offsets.

Discovery of the offsets allows us to use *dd* to perform manual filesystem extraction but removes much of the guesswork.

Extraction with Binwalk

Automatic extraction of files and filesystems
Bootloaders, filesystem images/disks
Individual files, directory reconstruction with file paths

```
$ binwalk --extract camera-firmware.bin
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC:
<output trimmed for brevity>
2097216       0x200040        Squashfs filesystem, little endian, version 4.0,
compression:xz, size: 3353204 bytes, 407 inodes, blocksize: 131072 bytes, created:
6225984       0x5F0040        JFFS2 filesystem, little endian

$ ls
camera-firmware.bin  _camera-firmware.bin.extracted
```


Extraction with Binwalk

If we add the *--extract* option to *Binwalk*, it will utilize the automatic component discovery, passing the offsets to dd in order to perform automatic extraction of all components. The resulting output is a directory ending in .extracted with all of the extracted components, ripe for our exploration.

This action will recover bootloaders filesystems and other components. If filesystems are discovered, the filesystems contents will also be extracted, and placed in the original directory structure from the recovered filesystem.

Binwalk Pro

Cloud-based, subscription driven version of Binwalk

- Supports open-source release and development
- Very affordable for additional features

Improved file detection, encryption, and filesystem handling

Solid investment if firmware lots of analysis is performed

Binwalk Pro is conspicuously absent after ReFirm Labs was purchased by Microsoft in June of 2021. We eagerly await its epic return!

Binwalk Pro

Binwalk Pro is a cloud hosted version of Binwalk based around a commercial subscription model. This subscription model with enhanced features, such as improved filesystem detection, automatic decryption, and better overall firmware file component detection, helps drive and support the open-source version.

It is the author's opinion that the overall cost of Binwalk Pro is quite affordable, especially if one spends a lot of time performing firmware analysis/.

Entropy Analysis

Entropy analysis can help reveal file types

Compressed images, encrypted files have a different distribution of characters

Automation of certificate discovery, ssh keys, LZMA, and ZIP compressed images

Entropy Analysis

Entropy analysis is a valuable tool for locating all sorts of fun parts within our firmware. By performing a statistical analysis of the character distribution encrypted, and compressed components tend to stand out! With normal ”text” there is a character distribution that has some overly high counts; think about chat game show where you pick words at the final challenge: N, R, S, T, L, E because they are used in higher distribution. However, in a compressed or encrypted format, there is an even distribution of characters. This distinct change makes it easy to identify those unusual/fun components, from ssh keys, compresses filesystems or images, and even webserver certificates.

Entropy Analysis with ENT

```
$ ent camera-firmware.bin
Entropy = 7.403248 bits per byte.

Optimum compression would reduce the size
of this 11075648 byte file by 7 percent.

Chi square distribution for 11075648 samples is 50333367.77, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 140.7984 (127.5 = random).
Monte Carlo value for Pi is 2.714056408 (error 13.61 percent).
Serial correlation coefficient is 0.350399 (totally uncorrelated = 0.0).
```

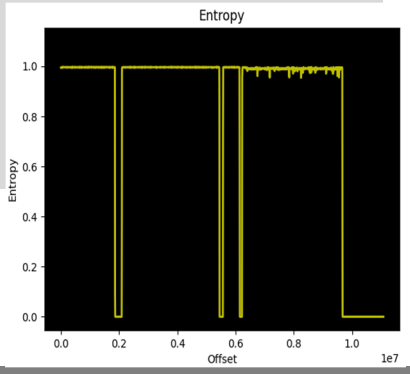

Entropy Analysis with ENT

ENT is a great tool for evaluating individual files for the statistical distribution of characters within the file. In this fashion it is great for helping to determine the presence of encryption or compression of overall firmware images or separated disk images. Unfortunately, *EN*T is not good at identifying individual files within an overall disk image.

Entropy Analysis with Binwalk

```
$ binwalk -E camera-firmware.bin

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Rising entropy edge (0.994618)
1859584       0x1C6000        Falling entropy edge (0.093710)
2097152       0x200000        Rising entropy edge (0.991917)
5451776       0x533000        Falling entropy edge (-0.000000)
5574656       0x551000        Rising entropy edge (0.996314)
6139904       0x5DB000        Falling entropy edge (0.638690)
6225920       0x5F0000        Rising entropy edge (0.956936)
9670656       0x939000        Falling entropy edge (0.349518)
```



Entropy Analysis with Binwalk

On the contrary to *ENT*, Binwalk can also perform entropy analysis, but by examining a filesystem or firmware images, noting the points in which entropy changes as we progress through the file. Binwalk when provided the -E flag will observe and record within our source file the points in which the entropy calculations change state. These positions can be provided to dd, in order to split out the parts with low entropy, indicating encrypted, compressed, or key content, so that we perform additional analysis, with other utilities such as *file* or a text editor such as *vi*.

QNX

Many analysis tools focus on Linux -based firmware.
- Binwalk, others can extract QNX filesystems so not all is lost!

QNX is Linux-like but with a microkernel.
RTOS-based operating system
Highly stable, but harder to implement
Commercial license, support from RIM/BlackBerry Limited
Filesystems can be Linux-like, or ETFS.
Extracted QNX filesystems have a different look and feel.
- The data is all there, just in a different layout.
- A little more hunting needed for what we expect under *nix

QNX

We spend much of our focus on Linux-based firmware, but there are many others out there, including the very common RTOS, QNX. Many of the tools we encounter do focus on Linux, largely because of the easy of entry, and the cost (free!) and familiarity with many implementers. QNX has a higher barrier to entry, in that it is commercially licensed, and is a quite complex build chain.

QNX can have Linux like filesystems (or ETFS, which is different altogether!), but is arguably more stable and features a microkernel (as opposed to the Linux monolithic kernel).

QNX is easily analyzed with Binwalk! The resulting output filesystems feel different, but all of the similar type of secret data we are often after is there. With a different layout to the filesystem than Linux, we just need to do more digging than normal.

More information on the QNX internals can be found at the QNX developers' site at http://www.qnx.com/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_sys_arch%2Ffsys.html.

QNX with Binwalk

```
$ binwalk --extract QNX-SUV-firmware.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------

10524882      0xA098D2        gzip compressed data, has 6494 bytes of extra data, last
modified: 2017-11-10 14:05:28
239486536     0xE464648       eCos RTOS string reference: "ecos.ans"
292262475     0x116B924B      MySQL ISAM index file Version 11
345045969     0x1490FBD1      MySQL ISAM compressed data file Version 3
400680632     0x17E1E6B8      QNX6 Super Block
451053654     0x1AE28856      MySQL ISAM index file Version 1
<output trimmed for brevity>...
```


QNX with Binwalk

In similar fashion to a Linux based firmware, Binwalk can do automatic detection of QNX firmware. In this example passing the --extract option to bin walk with a firmware, it can automatically extract QNX filesystems for us to explore.

Summary

Firmware analysis gives us a look at the OS of a device when we don't have physical hardware

Several tools for extraction, analysis, entropy analysis

Summary

In this module we examined the value in obtaining an offline copy of the operating software in the form of firmware either from downloaded copies or retrieved from a device from hardware hacking. We reviewed several tools for performing amazing analysis and deep dive and extraction of file system components.

Once extracted we can use our knowledge of *nix-based systems to explore. When we encounter QNX based systems, we have tools to extract, but we may be left needing to explore the unknown.

Exercise: Recovering Filesystems with Binwalk

Using your lab book, complete the exercise, *Recovering Filesystems with Binwalk*.

This page intentionally left blank.

Course Roadmap

Introduction to IoT Network Traffic and Web Services

Exploiting IoT Hardware Interfaces and Analyzing Firmware

Exploiting Wireless IoT: Wi-Fi, BLE, Zigbee, LoRa, and SDR

SECTION 2

1. **Background and Importance**
2. **Opening the Device**
3. **Examining and Identifying Components**
   Exercise: Obtaining and Analyzing Specification Sheets
4. **Discovering and Identifying Ports**
5. **A Soldering Primer**
6. **Sniffing, Interaction, and Exploitation of Hardware Ports**
   Exercise: Sniffing Serial and SPI
7. **Other Ways of Recovering Firmware**
   Exercise: Recovering Firmware from PCAP
8. **Firmware Analysis**
   Exercise: Recovering Filesystems with Binwalk
9. **Pillaging the Firmware**
   Exercise: Pillaging the Filesystem

This page intentionally left blank.

Pillaging the Firmware

With the firmware, we have filesystem(s)

Filesystems distributed far and wide reveal secrets!

We just need to investigate with some educated guesses

Every implementation will be different!

- In some cases, we know right where to look
- Others, we are left guessing

Files system digging combined with some Linux experience is invaluable

Pillaging the Firmware

Once we've been able to obtain and extract filesystems from our firmware, we need to explore to find all sorts of goodies. The problem with firmware is, that once it is distributed, with as a file, or on a device itself, these secrets are now "in the wind" and are there to be discovered by parties with enough time and willpower.

Drawing upon previous experience with Linux, and often how and where configuration files are stored, we can gain knowledge about the systems configuration and the secrets often required to keep them in operation. In most cases, we will know right where to look for the standard configuration file or do a little digging to track them down based on individual changes made by the manufacturer. Sometimes, especially when analyzing web applications, custom scripts, and custom applications we need to dig a lot deeper. Digging deeper may leave us with more questions than we answer, but with enough time and persistence, (and learning along the way) we can figure everything out.

Interesting Things to Look For

Having "physical" access to the disk image off device is almost as good as having access to a running device!

Physical access is like having full access.

We just need to look in the right places!

Some are "obvious", some not so much...

Access to a running system is always better for interaction. We may not have access, or access to the device: firmware analysis is the next best thing!

Interesting Things to Look For

If we can take an extracted file system from a firmware image and mount it much like we would a normal disk, or analyze the directory output from Binwalk, is almost as having access to a running device. While there will be no processes running, nor the ability to start them (without some forays into emulation) it effectively gives us full access to the files system as if we were sitting in front of a console of the devices.

We just need to think logically about where we need to explore on the filesystem based on what we know about the manufacturer, device functionality, and observed software from both use and network reconnaissance. With these observations we can make educated guesses as to the types of things we should look for!

Firmware analysis is incredibly valuable when we cannot get access to a running system for interaction. A running system, or a firmware image of one used in the wild are our best scenarios for finding the most issues, recover real-world secrets, and observe the system after first startup post user interaction) setting of passwords, configuring Wi-Fi, etc.,) are the best, default firmware from the manufacturer is the next best thing. As an example, with default, unused firmware, we may not be able to observe user configured Wi-Fi networks and their stored secrets; however, we should be able to examine default partially empty configurations and follow the interactions and tools that are used to create the storage of the credentials. It will take more work to connect the parts, and in some cases, we may be left guessing.

/etc/shadow, /etc/passwd

/etc/shadow and /etc/passwd are sources for user accounts and password hashes

- /etc/shadow should be restricted from read by a regular user
- Limited information in /etc/passwd

Undocumented backdoor accounts?

Offline password cracking with JtR, Hashcat

Hacking like it is 1980: Many IoT devices use world readable /etc/passwd for password hash storage.

/etc/shadow, /etc/passwd

In traditional, modern Linux systems, even those used on IoT devices, we find user account password hash storage divided between /etc/passwd and /etc/shadow. While /etc/passwd is world readable by anyone, on running systems /etc/shadow should be restricted for read by only root users; /etc/shadow are where the super-secret user password hashes are stored!

This is a great place to begin our investigation, especially if we are doing "offline" analysis of firmware as this can reveal user credentials for gaining access to love systems or even observing backdoor accounts. Of course, the passwords here should be hashed, in /et]c/shadow, but with unrestricted access to the file system, the traditional protections don't apply. We can capture those hashes and perform some password cracking, with ether a dictionary-based attack with *JtR*, or with some liberal application of brute force with *hashcat.*

To prove a point about the disturbing approach to security often found in IoT devices, this author has encountered several occasions recently where the user passwords were stored in the /etc/passwd file. Traditional Unix systems haven't used this approach in decades, and the implementers likely had to change the configuration of the system intentionally to make it weaker!

JtR

Using /etc/shadow and /etc/passwd together to create hashlist for JtR with *unshadow*.

```
$ unshadow passwd shadow > passwords.txt
$ john --wordlist=/usr/share/wordlists/rockyou.txt passwords.txt
<trimmed for brevity>
Remaining 2 password hash
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:51 DONE (2018-08-06 14:10) 0g/s 27478p/s 27478c/s 27478C/s
Zzzzzzz1..zzzzzzzzzzzzzz
Session completed
$ john --show passwords.txt
root:admin:0:0:sys:/dev:/bin/sh
reboot:reboot:1:1:sys:/bin/reboot
```


JtR

In this example we are using JtR to perform a dictionary-based attack against some recovered password hashes. In the first step aw need to use *unshadow* to combine out /etc/passwd and /etc/shadow files bac into one monolithic file for use with JtR. In the case of poorly configured systems, with a single /etc/passwd file (and no /etc/shadow), the use of unshadow is not needed.

Hashcat

```
$ hashcat -m 500 -a 0 -o cracked.txt shadow
/usr/share/wordlists/rockyou.txt
<trimmed for brevity>
[s]tatus [p]ause [b]ypass [c]heckpoint [q]uit => s
Session..........: hashcat
Status...........: Running
Hash.Type........: md5crypt, MD5 (Unix), Cisco-IOS $1$ (MD5)
Hash.Target......: hashes.txt
Time.Started.....: Mon Aug  6 14:18:10 2018 (30 secs)
Time.Estimated...: Mon Aug  6 14:21:08 2018 (2 mins, 28 secs)
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.Dev.#1.....:    15816 H/s (7.84ms) @ Accel:256 Loops:125
Recovered........: 5/7 (71.43%) Digests, 5/7 (71.43%) Salts
Progress.........: 1648627/9845703 (16.74%)
Rejected.........: 3059/1648627 (0.19%)
Restore.Point....: 234929/1406529 (16.70%)
Candidates.#1....: 9dH8eJEs -> 9notenler
HWMon.Dev.#1.....: N/A
```

MD5CRYPT?

/etc/shadow and /etc/passwd could use one of several types. We've chosen md5crypt (type 500) but could also be types 3200, 7400 or, 1800.

Hashcat

Alternatively, we can perform wordlist-based attacks with hashcat (or even brute force and mask-based attacks, discussed earlier in this course). In this case, we are using the -o flag to specify the output file of *cracked.txt* and our input file of *shadow*.

Unix password hash methods can take several different formats, and here we're chosen type 500 for mdcrypt. We can determine the hashing type by examining the format of the stored hashes; /etc/shadow passwords are stored in the format of idsalt$hashed; the $id is the algorithm for storing the password:

1 is MD5 or hashcat type 500

$2a$, $2y$ is Blowfish or hashcat type 3200

5 is SHA-256 or hashcat type 7400

6 is SHA-512 or hashcat type 1800

/var/www Contents

With traditional Apache and nginx installs /var/www is the default location for webserver content, but large footprint for IoT.

Many deployments utilize built-in *busybox* for simplicity.

Both methods deliver web content from disk.

- But from where on disk?

Examining the content offline can yield remote attack opportunities.

Depending on the device, we may be left hunting through the file system for web interface content.

/var/www Contents

With access to the file system, we can gain access to the contents and scripts for any web-based configuration utilities. On full Unix systems with Apache or Nginx the webserver content is usually found in /var/www, but the application of these webserver technologies if often too large of a footprint for our limited processing power and capacity IoT devices. Typically, what we find instead of a traditional webserver server software is the highly integrated, monolithic *busybox*, which is significantly smaller, and performs the function of many common Unix utilities.

While a small webserver is great for implementation in IoT, it does mean that the implementer can specify their own unique path for pwbserver content, if not in /var/www. By doing some quick filesystem searches, we should be able to locate the web content if in a custom place somewhere on disk.

Once we've located, it we can analyze all of the web and script content for implementation failures.

Webserver Content Analysis

Analyzing web server content can reveal

- Web page names for attempting authentication bypass.
- Opportunities for remote command injection.
- Observation of authentication mechanisms.
- Device functionality without authentication.
- Backdoor accounts.
- Failure in device settings/application.
- Commands being sent to other components/custom applications.
- cgi-bin scripts.

Depending on firmware, the ability to modify/replace content

Webserver Content Analysis
Once discovered, analyzing the webserver content can lead us to several implementation failures:

- Web page names for attempting authentication bypass – by observing the filenames that are used for configuration, we can than attempt to access these pages prior to authentication. Typically, the filenames would not be revealed to us until after authentication.
- Opportunities for remote command injection – by examining how user submissions are handled, we can determine if it is unfiltered and how it may be passed to Unix commands, executing our own commands too.
- Observation of authentication mechanisms – with access to the authentication mechanism we can analyze it for failures and other processing. This may require some RE of custom binaries.
- Device functionality without authentication – by examining the contents of the page contents, we can determine capabilities and features of the device without actually interacting with one where we may not possess credentials.
- Backdoor accounts- in combination with analysis of the authentication mechanisms, this may lead us to discover hidden, backdoor accounts, outside of the normal authentication channels.
- Failure in device settings/application – finding custom apps and observing their interactions can reveal implementation failures.
- Commands being sent to other components/custom applications – Observing of commands “hidden” in web page code can reveal functionality and use of custom manufacturer tools, possibly providing other avenues of attack or command injection.
- cgi-bin scripts – overall scripting for command injection, use of the underlying OS and custom binaries, or even hard coded credentials can be used to expand the attack surface, wither with command line access, or with access to the web interface.

Depending on the type of file system we are interacting with, it may be possible to replace or modify webserver content and repackage it back into a good firmware, allowing us to obtain additional access.

System Configuration in/etc.

The /etc directory contains configuration for many system services and functionality.

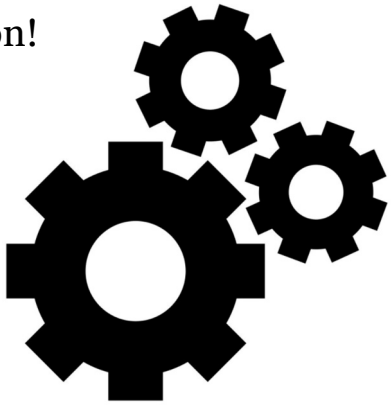
A wealth of information on system configuration!

DHCP, DNS, system components

System authentication, webserver, SSH/Telnet

Wi-Fi networks, passwords

Other wireless configuration(s)



System Configuration in/etc.

The overall set of system configuration files in the .etc., directory can provide a wealth of information about the operation and installed/used components of the system.

In a number of cases, the author has observed unusual settings in the .etc.directory for system components, such as hardcoded DNS servers, additional DHCP scopes, and even configuration and services enabled on unusual ports for webservers, SSH and Telnet with alternative authentication mechanisms!

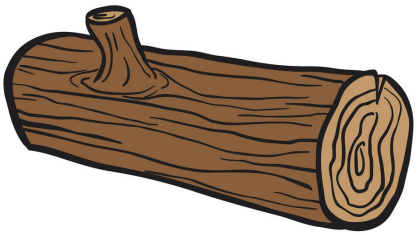
Additionally, configuration and keys for Wi-Fi and other wireless networks are also stored in plaintext configuration in the /etc/directory as well. Recovering these credentials can expose those wireless networks to additional, network-based attacks.

System Logs

System logging is great for system status.

- Good status for operating services and authentication
- Great when things fail
- Even better when too much logging is enabled

Don't forget output from *dmesg!*



System logs work best from a running, active system, or firmware acquired from one!

System Logs

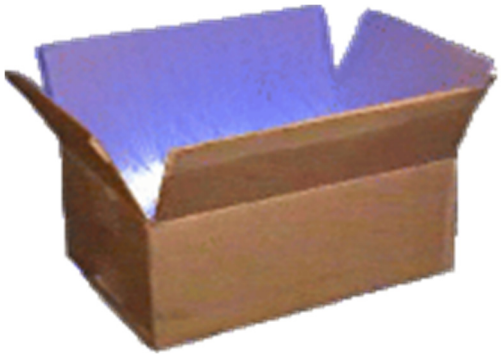
While often small due to limited storage space, system logging output can be a wealth of information. It will often tell us about authentication attempts on systems and other services in either syslog output or that from dmesg. If nothing else, it can give us information about the running system, but best when it tells us about things failing, or when the logging is overly verbose. It has been this author's experience observing verbose debug log messages from custom device apps, sending all sorts if useful data in plaintext, including connection strings, usernames and passwords, and the application that raised the offending error.

Most manufacturers sanitize and remove logs before creating the default set of firmware. This makes logs from default firmware useless. Logs are best observed on a running, actively-used system.

Webserver Configuration

Much of the configuration of IoT devices is done via a web browser. What is the configuration of the webserver?

- Filesystem paths
- Authentication? Basic auth?
- .htaccess?
- cgi-bin? Configured languages?



Webserver Configuration

The webserver implementation and configuration cash offer us a host of additional details, including that darned location of the webserver content directory! Also, it can tell us about the enabled languages used for cgi scripts (php, nodejs, etc.,) which could allow us to more accurately target attacks and find implementation issues.

Finally, observing some configuration components, we can more accurately determine authentication types, and or obtain access to additional credentials and file restrictions though examination of *.htaccess* files.

Private Certificates

Private webserver certificates are stored on the device.

- We have access to the device.

Are the certificates the same across every installed version of the firmware

- Generated at creation time?

Are they unique per device, created on first boot?

Possession of webserver private certs provides for MiTM and traffic decryption opportunities.

Private Certificates

If we consider the webserver configuration, most of the modern web interfaces with implemented with HTTPS, which requires some public/private certificate pairs. With access to the firmware, we may be able to recover the private certificate, allowing us to decrypt traffic to and from the webserver. This may not be an issue right away, as we should be able to capture our own traffic in plaintext from our system. But what if we could decrypt someone else's? Or use that certificate chain on own web server and act as a proxy to capture credentials.

Depending on the implementation the certificates may be static and distributed with the same one across every firmware install. Shared keys across devices in the public across diverse organizations is not a good thing. A better choice here would be to have unique keys which is hard to implement at time of manufacture, but easy to do as part of an initial boot process, when installed by the end user.

SSH Keys

User .ssh directory contains SSH configuration, public, and private keys

Public keys great for placing on other devices for authentication

Authentication requires the private key

Are the SSH keys unique, generated on boot?

Many devices fail to remove the private key, granting access.

Shodan can find similar devices where SSH authentication may work using the recovered private keys.

SSH Keys
In similar fashion to webserver certificates, we can run into the same problem with SSH public/private keypairs. These keypairs can be more readily be abused for authentication across hundreds of devices, especially if key trust was implemented by the manufacturer. Again, here it is best for the manufacturer to generate unique keys on boot, but many times the fail to do this and implement static keys.

Custom Scripts/Apps

Custom content is where many mistakes are made.

Extra functionality on the OS, privilege

- Looking for sudo bits!

Direct control of hardware components

Scripts as text are easy to parse, dissect

Custom apps require an operating environment or RE.

RE can reveal secrets, but no interaction with real hardware

Custom Scripts/Apps

The hardest parts for us to analyze are the individual custom scripts and binaries included by the manufacturer. These scripts and binaries are often used to control the hardware in some fashion and are custom to the device. The scripts are easier to follow along with and determine functionality so that we can abuse the functionality of the hardware, the binaries become much more difficult.

We won't have the source code for the binaries, (unless the developers are embarrassingly sloppy) so we would need to perform reverse engineering with IDA or Ghidra. This is not an easy task without deep, specialized knowledge.

While we are at it, we should look for tools that escalate privilege or have the sudo bits set, so that if we can access them through other means, say the web interface we can have complete control of the device.

Other Firmware?

Individual chipsets with unique instruction sets, storage

ESP8266, Arduino, TI CC430, ST72324 ...

Many of these left to IDA Pro, Ghidra for deep RE

Conversion from hex to binary formats may be needed for processing

- objcopy converts from Intel hex to binary for use with other tools as needed.

NWS LMS6 radiosonde with an ST72324J6T6 processor, identified as CPU type ST7->ST72324J6 in IDA Pro analyzing the hexfile directly.

```
$ objcopy -I ihex -O binary radiosonde.hex radiosonde.bin
```


Other Firmware?

What about all of the rest? We discussed Linux and QNX, but there are hundreds of different chipsets that use a multitude of RTOS systems of various flavors. The disparate device chipsets have different IDEs, languages, language interpreters and the list goes on. We are left to having to determine the platform and do reverse engineering with much more esoteric knowledge.

We can use our hardware hacking techniques to recover firmware, but we may need to perform some image format conversion with *objcopy* to make the result compatible without reverse engineering tools.

Summary

Pillaging the filesystem can reveal security failures.

- Some easy to find, others require time, exploration and *nix familiarity
- QNX is only a little bit different

Unix system experience of often very helpful.

Non-Linux or QNX firmware requires a really deep dive.

Summary

In this module we examined the value in digging into the operating system images. Once extracted we can use our knowledge of *nix-based systems to explore. Observing the various configuration files and software implementations can reveal configuration issues and opportunity for exploitation.

When we encounter QNX based systems, we have tools to extract, but we may be left needing to explore the unknown.

Finally, we reviewed other IoT systems that are not Unix/Linux or QNX based, which can vary widely in their format depending on the hundreds of possible platforms. These systems will require some deep dives, as the RTOS based systems often require analysis with binary reverse engineering tools.

Exercise: Pillaging the Filesystem

Using your lab book, complete the exercise, *Pillaging the Filesystem*.

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT

Larry Pesce
larry.pesce.556@gmail.com
Twitter: @haxorthematrix

James Leyte-Vidal
jameslvsec556@gmail.com
Twitter: @jamesleytevidal

Steven Walbroehl
steven.walbroehl@halborn.com
Twitter: @HalbornSteve

SANS INSTITUTE

11200 Rockville Pike
Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES

pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL

GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.