It's assembler, Jim, *but not as we know it*!

Morgan Gangwere
DEF CON 26

Hello, I'm Morgan Gangwere and this is *It's assembler, Jim, but not as we know it!* We're going to be looking at some fun shenanigans you can have with embedded devices running Linux (and other things)



Dedication

Never forget the
shoulders you stand on.

Thanks, Dad

First a shout-out to my father. He's an awesome man who taught me the importance of understanding tools and then making your own. This is him using a chisel with just one hand – and a tool.

whoami

Hoopy Frood

Been fiddling with Linux SOCs since I fiddled with an old TS-7200 EmbeddedARM board

I've used ARM for a lot of services: IRC, web hosting, etc.

I've built CyanogenMod/LineageOS, custom ARM images, etc.



So who am I?
I'm the hoopiest frood that ever did cross the galaxy, been doing Linux-y stuff since I was a wee lad making off with one of my father's dev kits, and I've built my own tools for some time.

A word of note

There are few concrete examples in this talk. I'm sorry.

This sort of work is

- One part science
- One part estimation
- Dash of bitter feelings towards others
- Hint of "What the *fuck* was that EE thinking?"

A lot comes from experience. I can point the way, but I cannot tell the future.

There's a lot of seemingly random things. Trust me, It'll make sense.

ARMed to the teeth

From the BBC to your home.

So let’s talk about ARM.

Short history of ARM

- Originally the Acorn RISC machine
- Built for the BBC Micro!
- Acorn changed hands and became ARM Holdings
- Acorn/ARM has never cut silicon!
- Fun fact: Intel has produced ARM-based chips (StrongARM and XScale) and still sometimes does!
- The ISA hasn't changed *all that much.*



ARM, the Acorn RISC Machine, was built for the BBC Micro in 1985 by Acorn Computer, predominantly designed by the hands of Sophie Wilson. Later, ARM was changed to Advanced RISC machine, and later into simply ARM. The ARM ISA has stayed the same since ARM2.. Mostly. There's been changes to keep up with the time, but you could read ARM assembler from the 80s and understand it today.



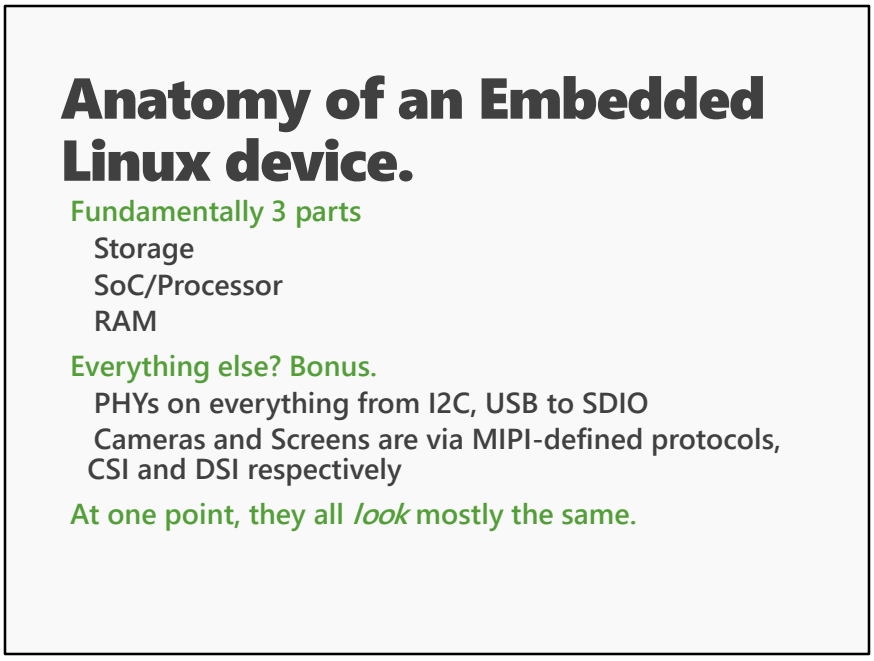
Network appliances, phones and routers have been running on top of ARM for quite a while: It's cheap, low power and versatile enough to do what most people want it to do. These small linux-based ARM devices have become fixtures in our houses and in enterprise. We're now seeing ARM-based laptops, with ASUS' NovaGo coming to market and several other devices based on Qualcomm's Snapdragon processor line running full desktop OS's on them and getting days of battery life.



The whole line of IKEA’s TRADFRI Smart LED lighting solutions run on ARM Cortex M0 chips, including the dimmers and bulbs. They go for months on a CR2032 battery.

Embedded Linux 101

So let’s talk about Embedded Linux devices in general.



Embedded Linux devices consist of three general parts: An SoC, Storage, and RAM. Most everything else is just a peripheral of some kind, like displays, USB devices, SDIO and more.

What is an SoC?

Several major vendors:
- Allwinner, Rockchip (China)
- Atheros, TI, Apple (US)
- Samsung (Korea)

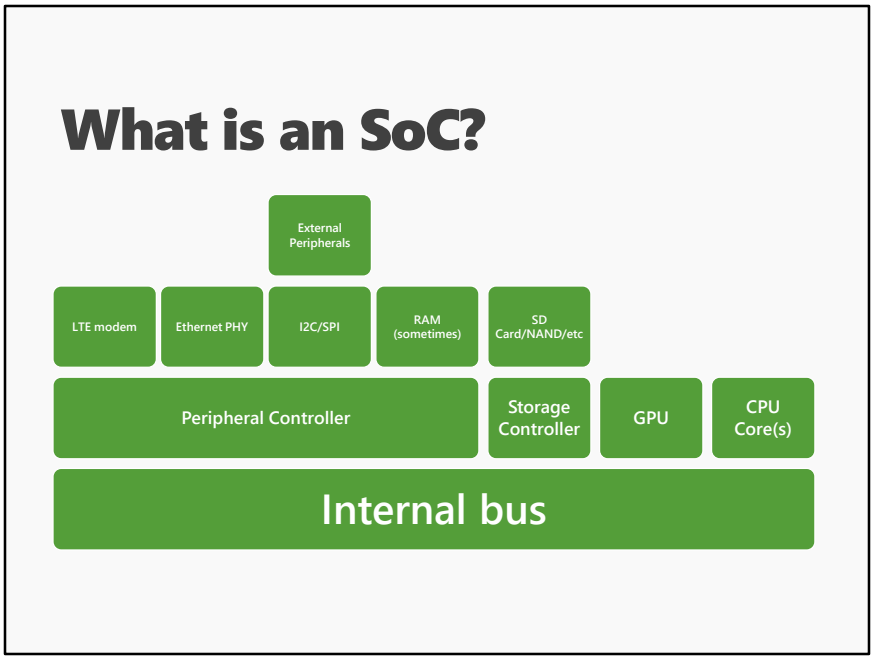
80-100% of the peripherals and possibly storage is right there on die

Becomes a "just add peripherals" design
- Some vendors include SoCs as a part of other devices, such as TI's line of DSPs with an ARM SoC used for video production hardware and the like.

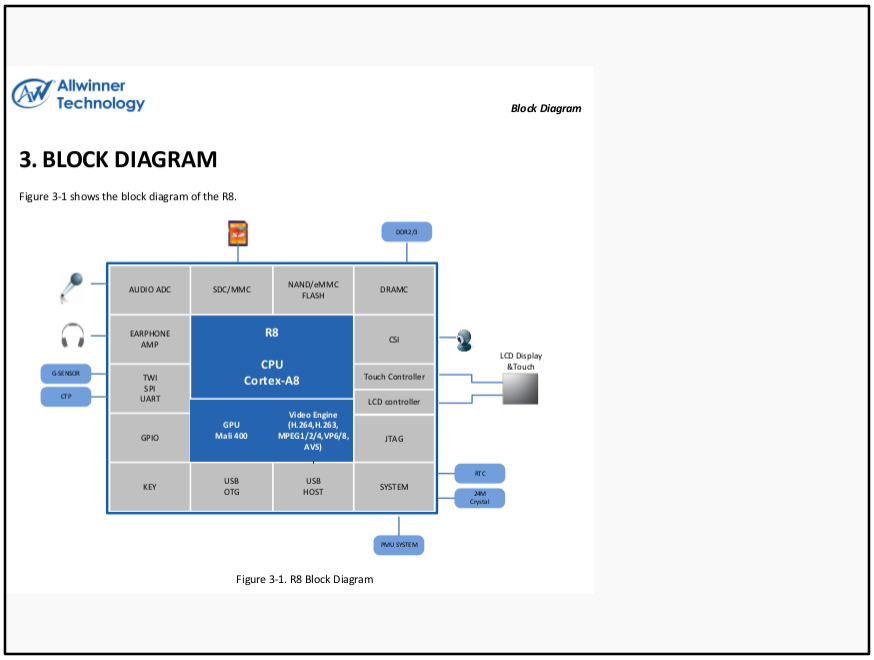
In some devices, there may be multiple SoCs: A whole line of Cisco-owned Linux-based teleconferencing hardware has big banks of SoCs from TI doing video processing on the fly alongside a DSP.



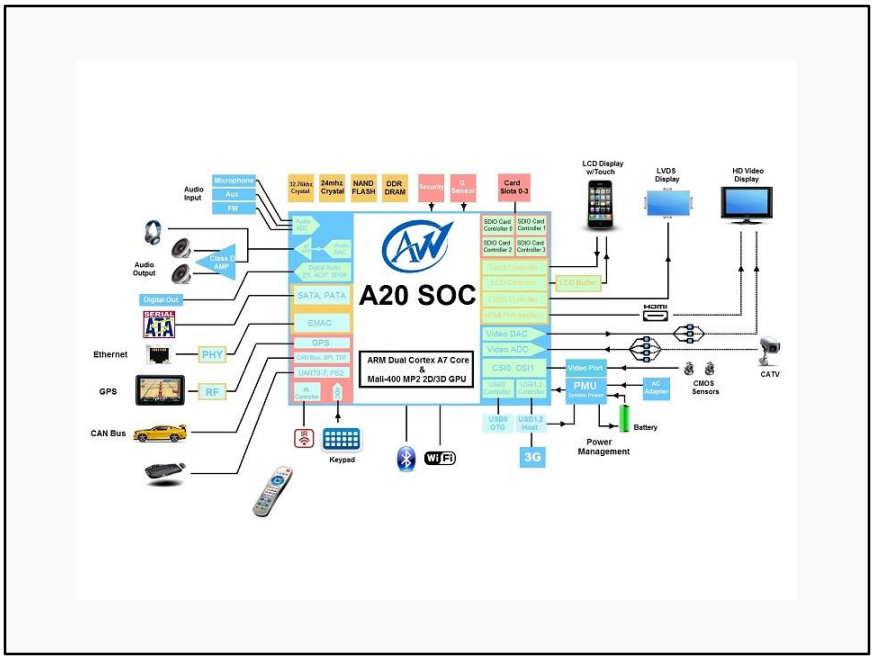
For those not familiar, an SoC, or System on Chip, is a "Just Add Peripherals" building-block in the design of devices. There's an SoC in your smartphone, for instance, that handles most of your phone's peripherals, including radios and display adapters. There's a handful of major players in the field, including several in the US, China, and Korea. SoCs are often bundled with application-specific silicon, such as a whole line of TI DSP chips which have an ARM SoC on-die to control it, complete with Ethernet PHY and a little bit of RAM.



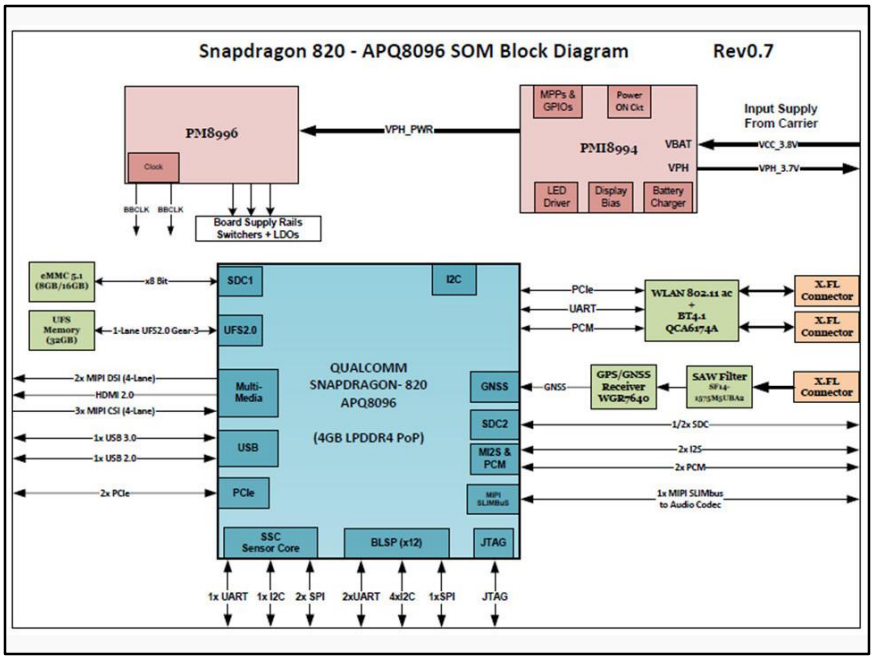
All the different SoC designs look astoundingly similar, so here's a really generic look at it. Everything rides on an internal bus of some kind, with peripherals sitting on some kind of controller. There's sometimes a secondary storage controller to make external storage such as SATA, NAND, or other storage technologies available early on or through some kind of abstraction layer.



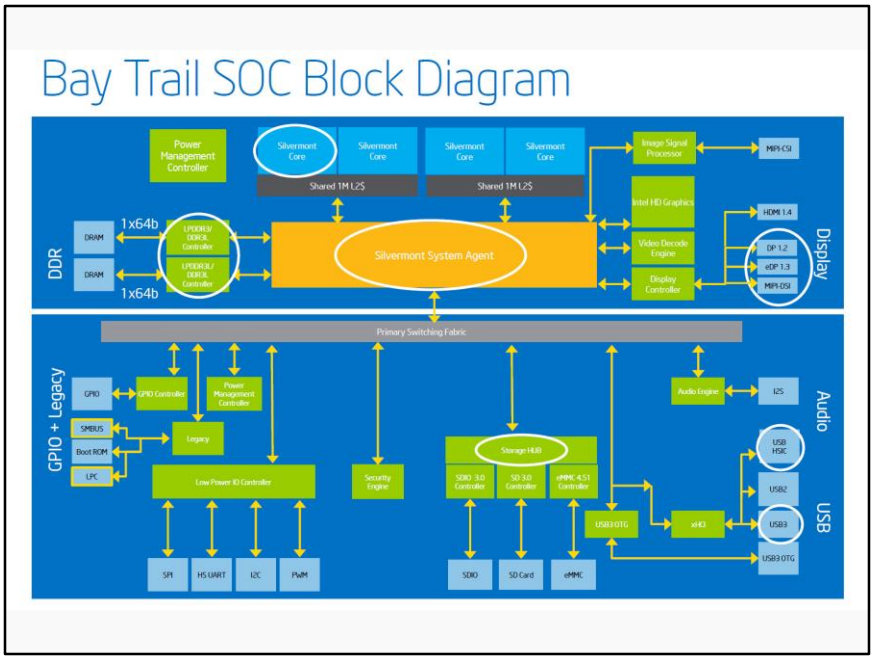
Once you've seen one



You've seen them all



They only really differ in the specific features: Here we see that this Snapdragon has 4G of LPDDR4 stacked on top, but also has 2 PCIe lanes!



Even Intel makes SoCs! This is a Bay Trail SoC – I suspect similar to what's in the MinnowBoard or something similar. As you can see, it even includes *Legacy* component support, such as SMBUS!

Storage

Two/Three common flavors

MTD (Memory Technology Device): Abstraction of flash pages to partitions

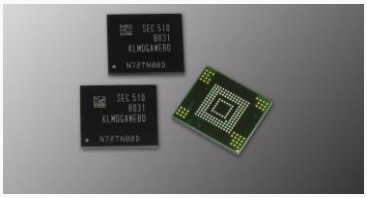


Storage is another story entirely. The first type of storage is MTD, or Memory Technology Device. This is NAND Flash at its core, and it gets abstracted out later by both the bootloader and Linux itself.

Storage

Two/Three common flavors

- MTD (Memory Technology Device): Abstraction of flash pages to partitions
- eMMC: Embedded MultiMedia Card



Then we have the venerable eMMC, which are a form of the now oldschool MMC, or MultiMedia Card, specifications.

Storage

Two/Three common flavors

- MTD (Memory Technology Device): Abstraction of flash pages to partitions
- eMMC: Embedded MultiMedia Card, SPI SD card
- SD cards



And now we have SD cards. Secure Digital has seen a few revisions of the form factor over time, but the protocol has stayed mostly the same. SD cards are almost 1:1 compatible with MMC cards and eMMC in a legacy, 4-bit mode that was used in The Old Days.

Storage

Two/Three common flavors

- MTD (Memory Technology Device): Abstraction of flash pages to partitions
- eMMC: Embedded MultiMedia Card, SPI SD card
- SD cards



However, seeing an exposed SD card is rare, but is a real jackpot opportunity. For example, the first generation Kindle uses it, but so does the Raspberry Pi.

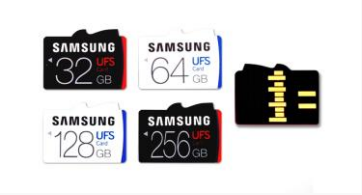
Storage

Two/Three common flavors

- MTD (Memory Technology Device): Abstraction of flash pages to partitions
- eMMC: Embedded MultiMedia Card, SPI SD card
- SD cards

Then there's UFS

- Introduced in 2011 for phones, cameras: High Bandwidth storage devices
- Uses a SCSI model, not eMMC's linear model



UFS is a new standard, somewhat common now in phones, for storing things. It's fast – Gigabits per second fast – and could very well become a standard for more devices in the future. It's also much more compatible with the classic SCSI way of looking at disks, which makes it ideal for things like Windows.

Variations

- Some devices have a small amount of onboard Flash for the bootloader
  - Commonly seen on phones, for the purposes of boostrapping everything else
- Every vendor has different tools for pushing bits to a device and *they all suck*.
  - Samsung has at least three for Android
  - Allwinner based devices can be placed into FEL boot mode
  - Fastboot on Android devices

There's some mix-and-match when it comes to storage though. There's often at least some baked-in flash that handles loading the bootloader, called the IPL, which is useful for bootstrapping everything else when push comes to shove. Every vendor has their own shitty way of cramming data onto the boot storage of the device, and they're all pretty bad.

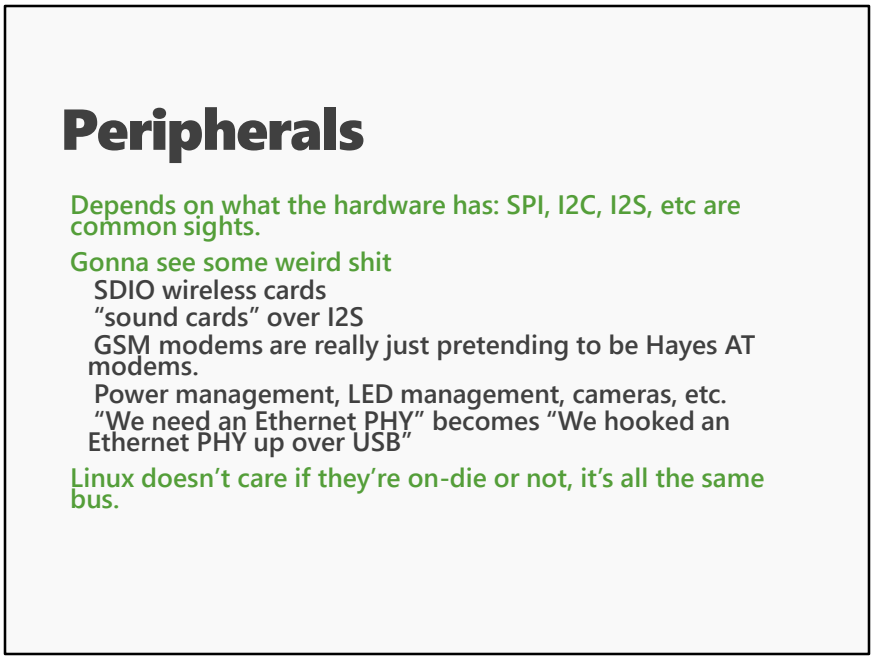
RAM

- The art of cramming a lot in a small place
- Vendors are seriously tight-assed
- Can you cram everything in 8MB? Some routers do.
  - The WRT54G had 8M of RAM, later 4M
  - Modern SoCs tend towards 1GB, phones 4-6G
- In pure flash storage, ramfs might be used to expand on-demand files (http content)



Let's talk about RAM.

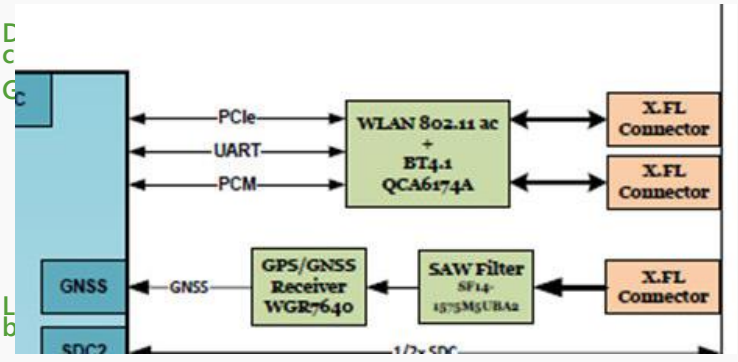
Sometimes, you get a lot of RAM – some phones are pushing 8 gigabytes of RAM just to hold Android. On the other hand, the WRT54G and a whole host of newer devices ship with 8MB of RAM. Talk about tight-assed.



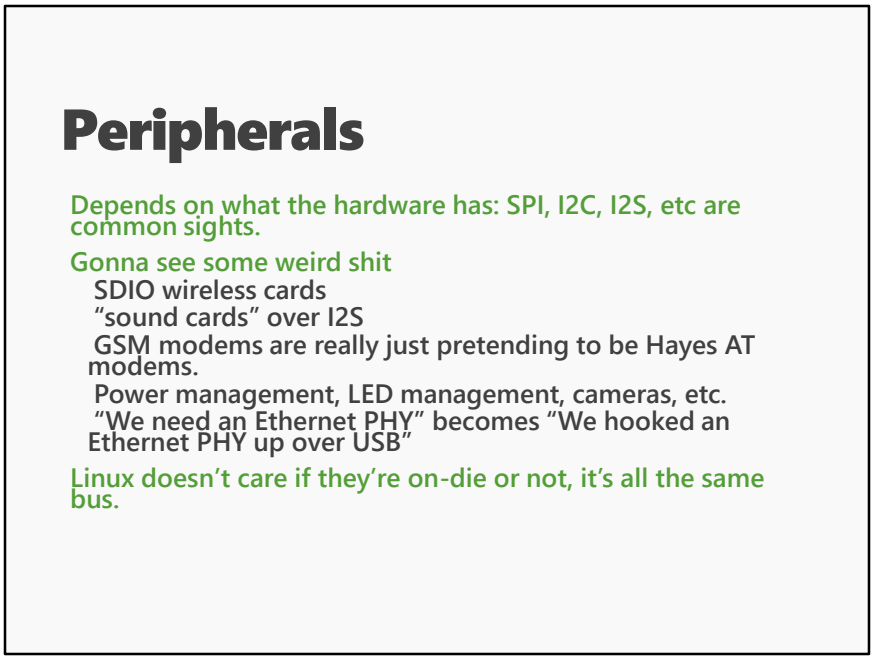
Now, on to peripherals. If your target application talks with these, you’re going to get nice and cozy with wiggling electrons.

This ultimately depends on what the SoC provides, which is a function of what the specific application needs. SPI, I2C and such are common. However, this hasn’t stopped some astoundingly dumb choices or seemingly weird choices.

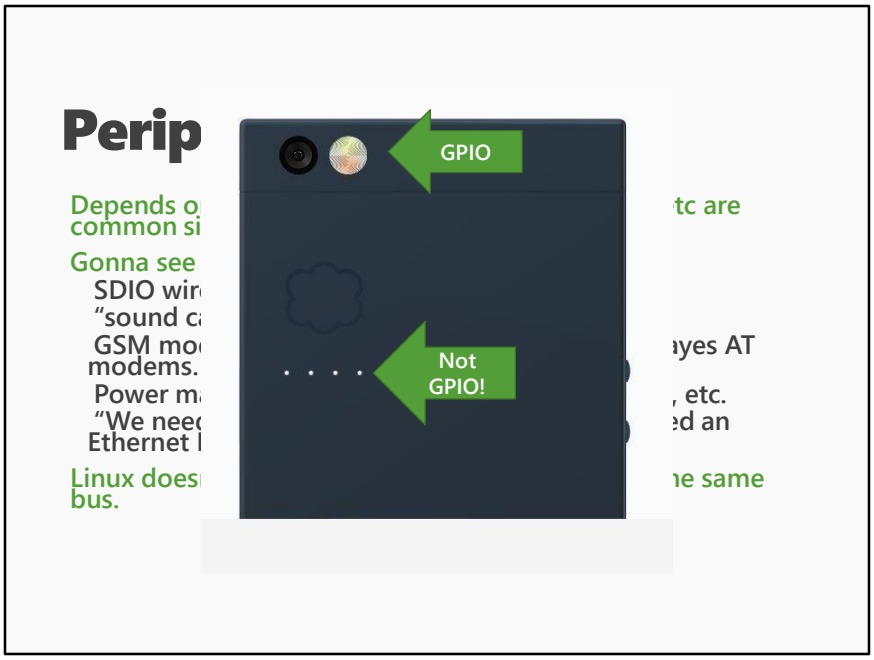
Peripherals



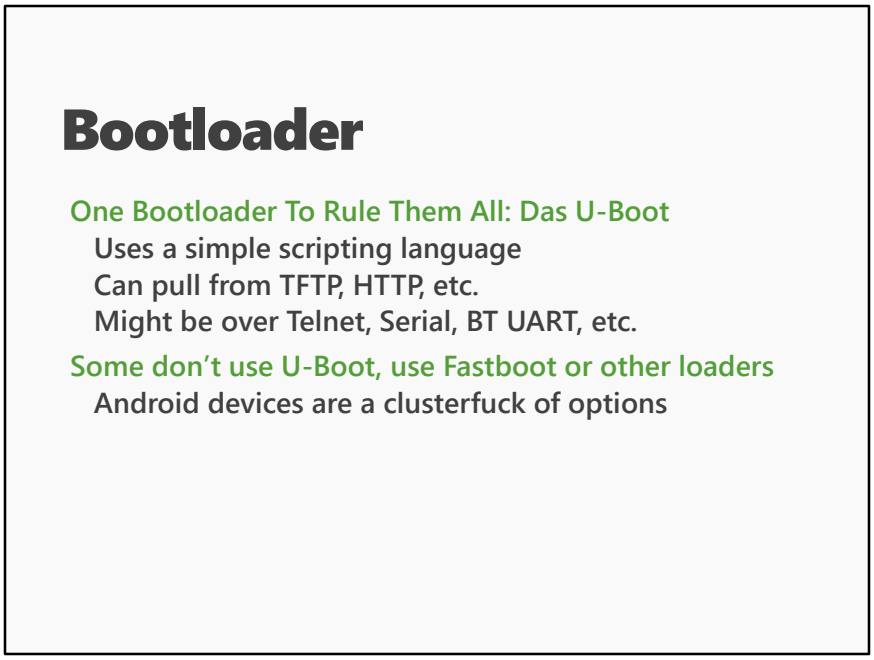
Remember that Snapdragon 820? It uses a PCIe lane, a UART and PCM channel to do WLAN and Bluetooth. Instead of cramming everything onto the PCIe lane, they chose to make it so that Bluetooth output and input are just lines on the internal audio codec.



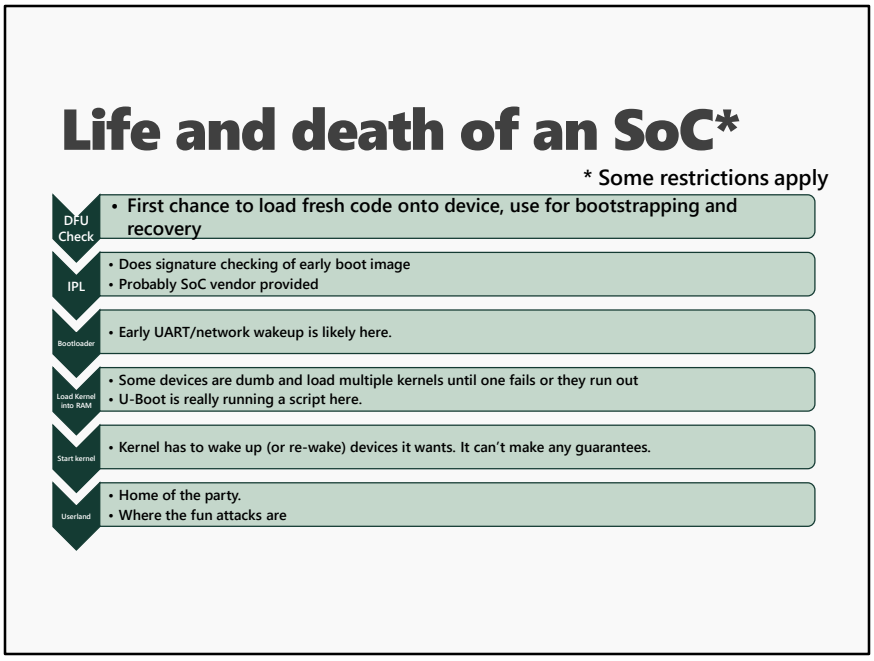
So many devices are going to be weird as fuck though. GSM modems for example are just Hayes AT modems with some extra glue.



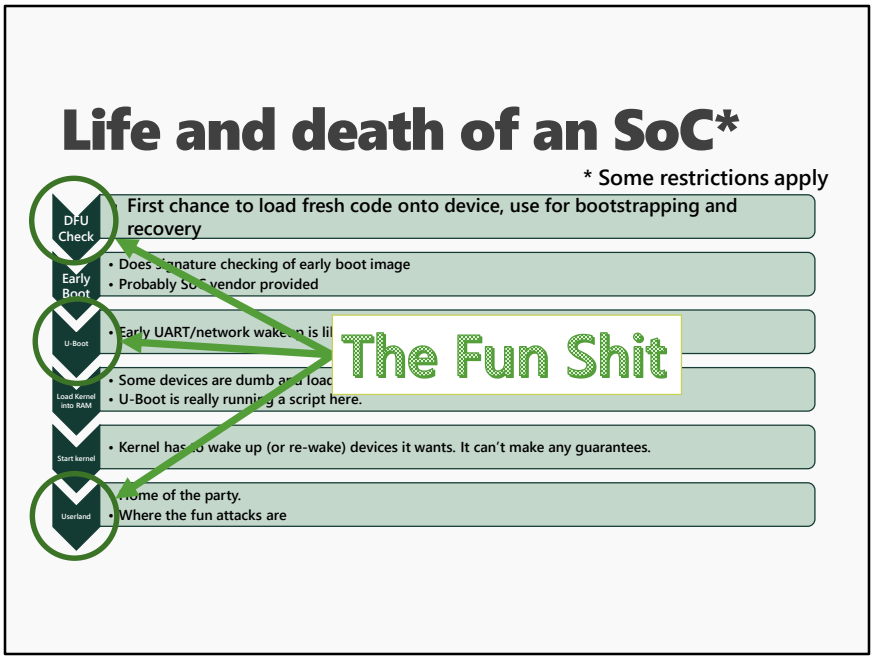
Sometimes, all the fancy LEDs on your device aren't GPIO pins on the SoC. Instead, they're an external peripheral. This is a Nextbit Robin, where the flash LED is a part of the camera hardware and the LEDs on the back are actually controlled by a TI LED controller that has its own tiny ISA.



Without the bootloader, you'd be nowhere, though. U-Boot is by far the most common bootloader for embedded devices, with routers and the like being the most common users. Phones and such are a whole different show, with a whole variety of options kicking around. Chances are, however, if it's not a phone it *probably* uses U-Boot.



The bootloader's job comes right after the IPL: Its job is to wake up any devices that haven't been as needed such as storage, load the kernel into memory, and kick it off. We've seen this dance before. This, as it turns out, is a whole lot cleaner on ARM than it is on Intel's clusterfuck of a design.



The real fun is when you can attack the earliest part of the boot sequence, the DFU or *Device Firmware Update* sequence. DFU mode is how fresh code is loaded on without any consent from the higher levels of the environment. This is also the place that most – if not every – OEM puts the most time into: Keeping someone else from fixing their problems becomes a steady stream of new devices. U-Boot and the userland are the other two fun parts of this, for the obvious reasons.

Root Filesystem: Home to All

A root filesystem contains the bare minimum to boot Linux: Any shared object libraries, binaries, or other content that is necessary for what that device is going to do

Fluid content that needs to be changed or which is going to be fetched regularly is often stored on a Ramdisk; this might be loaded during init's early startup from a tarball.

- This is a super common thing to miss because it's a tmpfs outside of /tmp
- this is a super common way of keeping / “small”
- This often leads to rootfs extractions via tar that seem "too big“

There are sometimes multiple root filesystems overlaid upon each other

- Android uses this to some extent: /system is where many things really are

Might be from NFS, might try NFS first, etc.

A device with no code to run is a device without a purpose. We need a root filesystem. Sometimes you get several, overlaid upon one another. Android does this and so does your LiveCD.

Attacking these devices

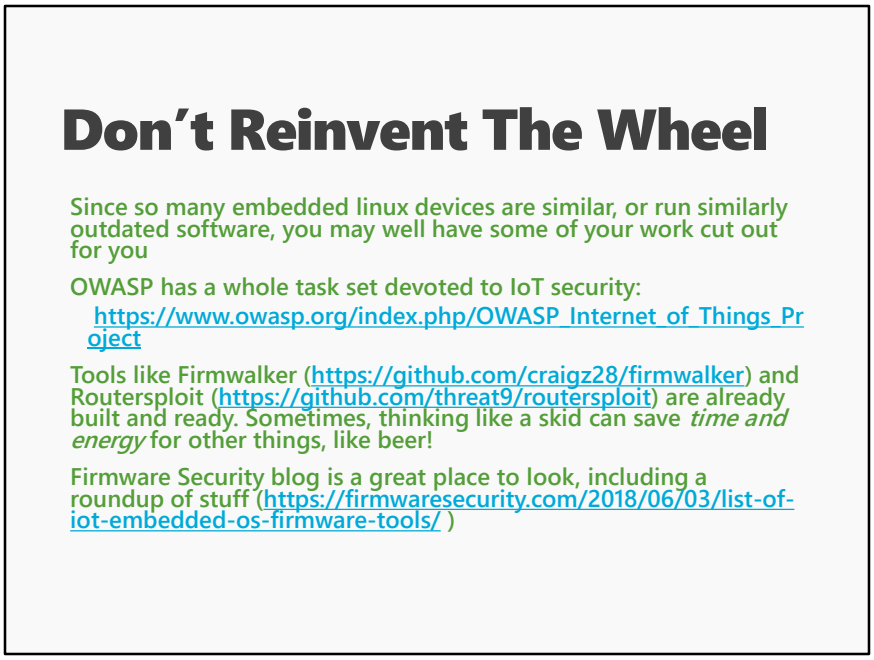
So how do we go about attacking these sorts of devices?

Step 0: Scope out your device

- Get to know what makes the device tick
  - Version of Linux
  - Rough software stack
  - Known vulnerabilities
  - Debug shells, backdoors, admin shells, etc.
- ARM executables are fairly generic
  - Kobo updates are very, VERY generic and the Kobo userland is *very aware of this*.
- Hardware vendors are lazy: many devices likely similar to kin-devices
  - Possibly able to find update for similar device by same OEM



Always. *Always* start with step 0: Get to know the device. Figure out what makes it go. Remember that ARM executables are really generic, and this is used by many vendors to ship the same code to different platforms. Vendors are lazy: they want to produce the most devices with the least amount of work possible.



Don't reinvent the wheel. Tools like Firmwalker and Routersploit, as well as the information on the OWASP IoT Security task set are meant to help streamline finding the information you need out of filesystems, updates, etc. If these sorts of things start really getting interesting, go read the roundups by the firmware Security blog.

Option 1: It's a UNIX system, I know this

If you can get a shell, sometimes just beating against your target can be fun

Limited to only what is on the target (or what you can get to the target)

Can feel a bit like going into the wild with a bowie knife and a jar full of your own piss

Debugger? Fuzzer? *Compiler?* What are those?



So, Option 1 is to treat it like any other UNIX system. Straight forward if you have a shell, you can start poking around. Often, however, you're already going to have a root shell or something very close to it, as well as only a handful of tools. You're not regularly going to get a debugger, compiler, even manpages. These devices are stripped to the bone.

Option 2: Black-Box it

Lots of fun once you're used to it or live service attacks.

Safe: Never directly exposes you to "secrets" (IP)

You don't have the bowie knife, just two jars of piss.



Option 2 is to black box the thing. This is turns your attack into any other straightforward service attack where you have no control over the device itself.

IANAL, but it would seem that this has less likelihood of you stumbling upon something vendor-secret, such as leftover binaries not pulled from the development process. .Again, IANAL. If you think you might possibly have an inkling that you need a lawyer, *get a fucking lawyer.*



These options fucking suck though.

It’s always better when you can investigate the whole goddamn binary.

Option 3: Reverse it

Pull out IDA/Radare

Grab a beer

Learn you a new ISA

The way of reversing IoT things that don't run Linux!

... but how the fuck do you get the binaries?



So the obvious answer is to reverse it. Already familiar with cracking open X86 executables in IDA? Radare? Time to learn a new ISA, grab a beer, and play the same game but with hot fresh binaries.

If your target executable talks with an external device that you can't emulate, this is probably your stop.

But *how the fuck do we get the binaries?* We'll get to that.

Yeah but I'm fucking lazy, asshole.

I don't want to learn IDA. I want to fuzz.

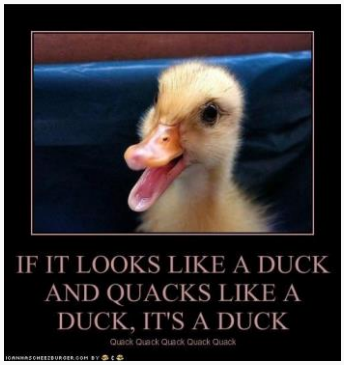
But I'm a lazy asshole, you say: I *need* that sweet sweet kernel debugger and AFL and the rest of my skid-cum-kernel-hacker tools!

Option 4: Emulate It

You have every tool at your disposal

- Hot damn is that a debugger?
- Oh shit waddup it's fuzzy boiii

Once again, *how the fuck do you get your binary of choice?*



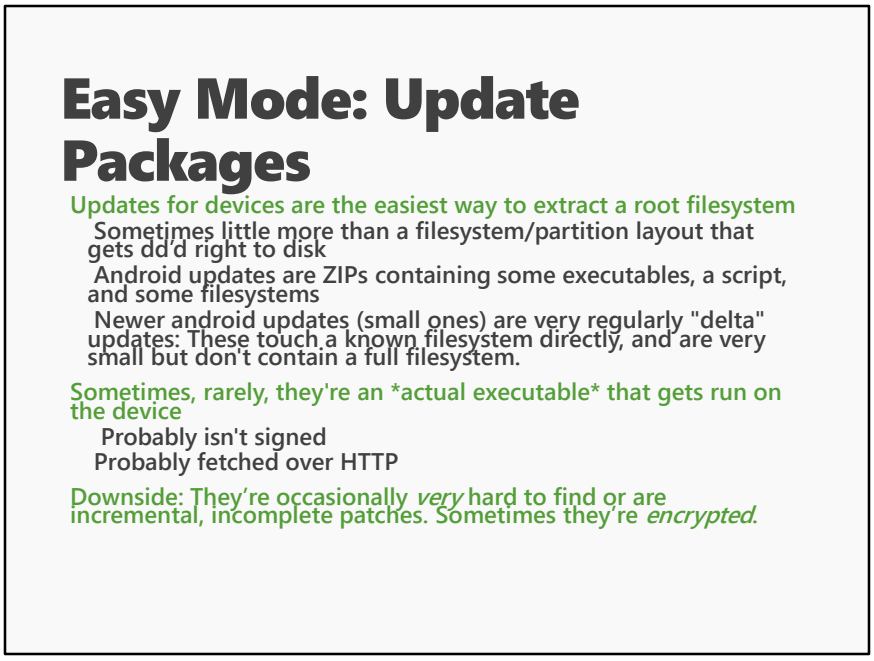
You emulate it! Emulation is a perfectly reasonable approach to many of these challenges. You've got all your normal tools in most cases, or you can at least cross-compile them.

But the issue still stands – how the *ever fuck* do you get your target binary?

Getting root(fs)

Let’s get root, then.

This can range from surprisingly easy to frustratingly hard, depending on the environment. Keep in mind rule 0 through this whole thing: Someone else has done something similar, probably. A few hours of googling and reading could save you many, many hours of head-scratching and anguished screams as you question the lineage of the engineers who designed a device.



So, easy mode is, most often, updates! Updates can be the most direct way to get the binaries of interest, especially if there's some new feature that's being rolled out. Often, these updates contain whole filesystems, but sometimes they're executables or binary patches against the blocks themselves. Sometimes, the bastards encrypt or obfuscate updates.

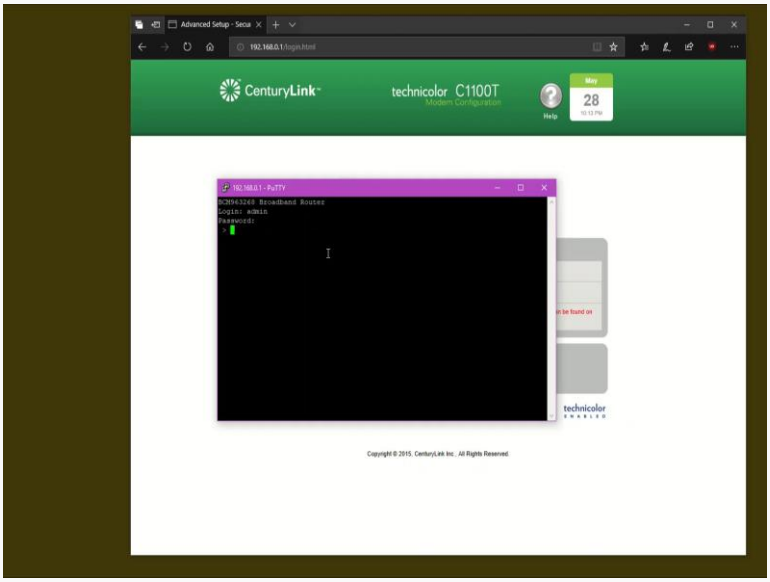
Medium: In-Vivo extraction

- You need a shell
  - Can you hijack an administrative interface?
  - Some ping functions can be hijacked into shells
  - Sometimes it's literally "telnet to the thing"
  - Refer to step 0 for more
- You need some kind of packer (cpio, tar, etc)
  - Find is a builtin for most busybox implementations.
- You need some way to put it somewhere (netcat, curl, etc)
- You might have an HTTPD to fall back on
- Need to do reconnaissance on your device
- Might need some creativity
  - Wireshark, Ettercap, Fiddler, etc

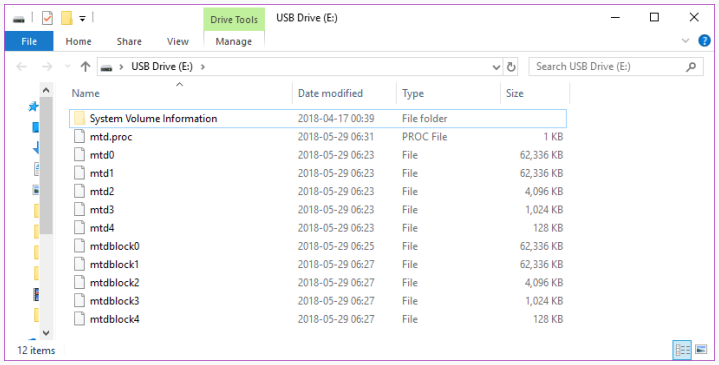
Next up is the Fuck it, we're doing it live of getting files off a device. This will require a certain amount of creativity on your part as you're living in a very barebones envinonment. Take the easiest way out possible.

Demo: Router firmware extraction (Actiontec Router)

So let’s see what this looks like.


Advanced Setup - Seca
192.168.0.1/login.html
CenturyLink
technicolor C1100T
Modem Configuration
Help
May
28
192.168.0.1 - PuTTY
BCM963268 Broadband Router
Login: admin
Password:
technicolor
Copyright © 2015. CenturyLink Inc. All Rights Reserved.

What did we get?

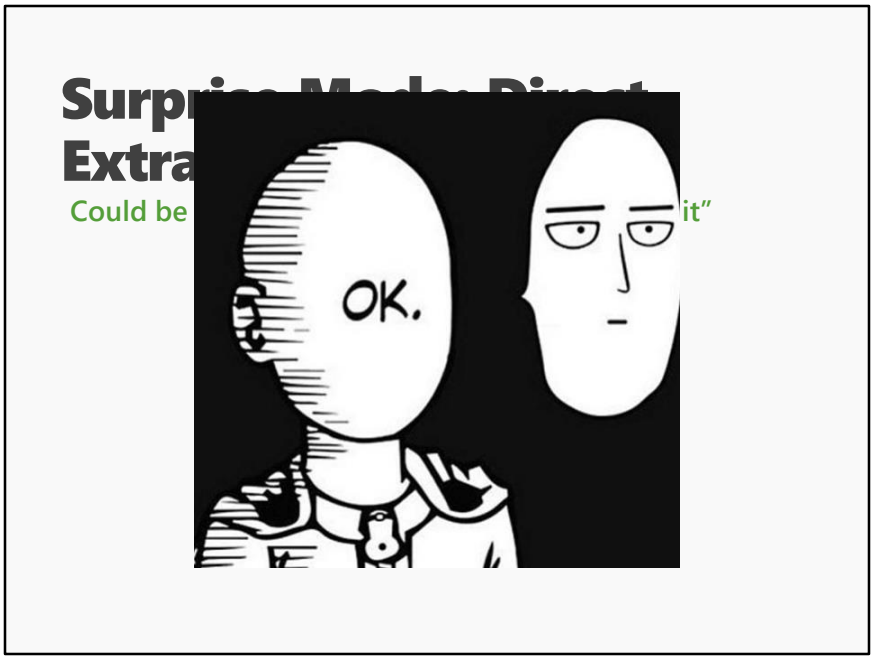


Hot damn, those look like filesystems.

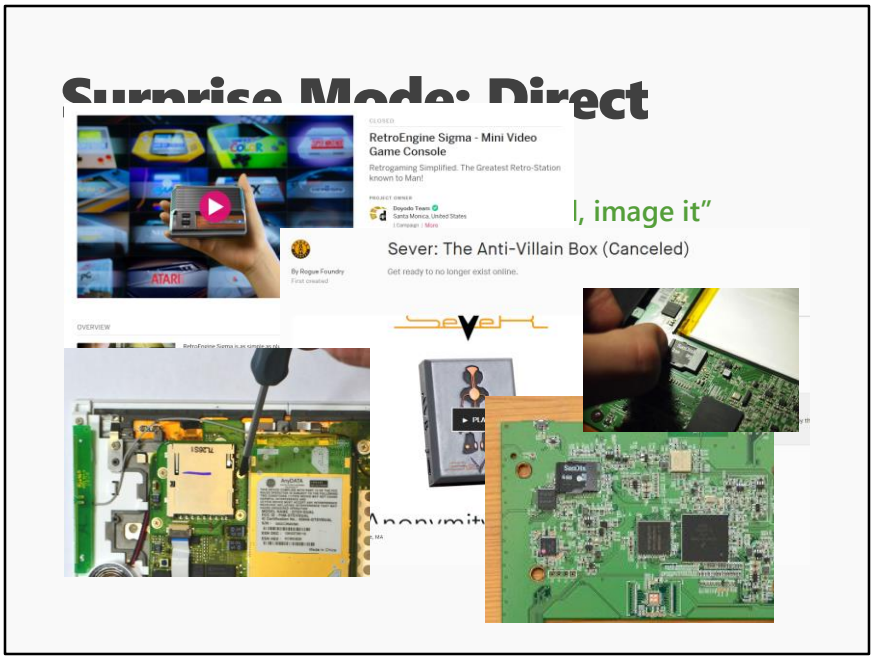
Surprise Mode: Direct Extraction

Could be as simple as “remove SD card, image it”

Next up is direct, complete extraction at the raw block level outside of the native environment. This could be as simple as pulling the SD card out and dumping that



And you're looking at me like "okay asshole, tell me something I don't know".

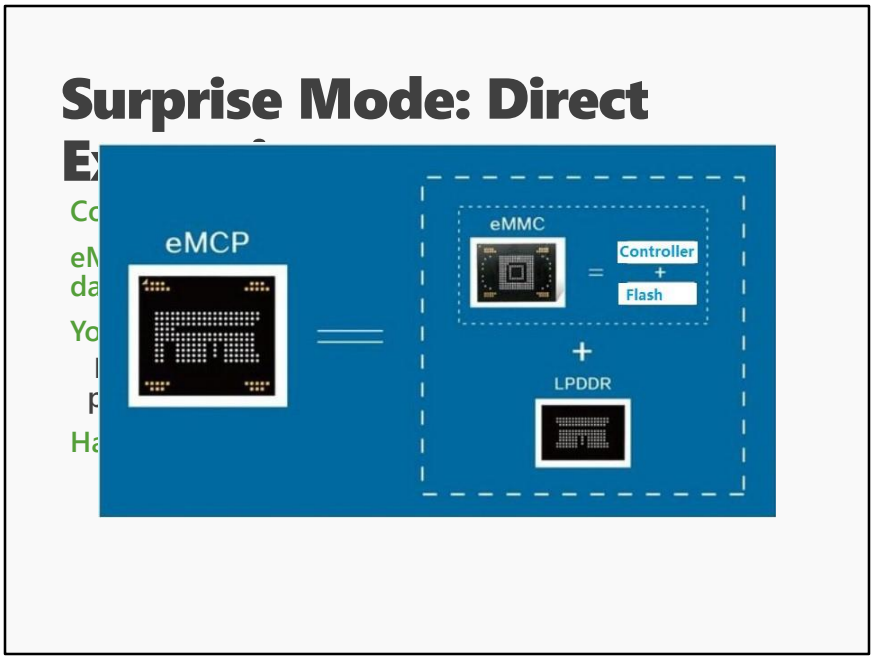


But the number of devices that have these is astoundingly large. These are all devices, a game console, an “anonymity device” a first generation Kindle and a Kobo H2O. Many of these devices are just a Raspberry Pi with some added hardware on top.

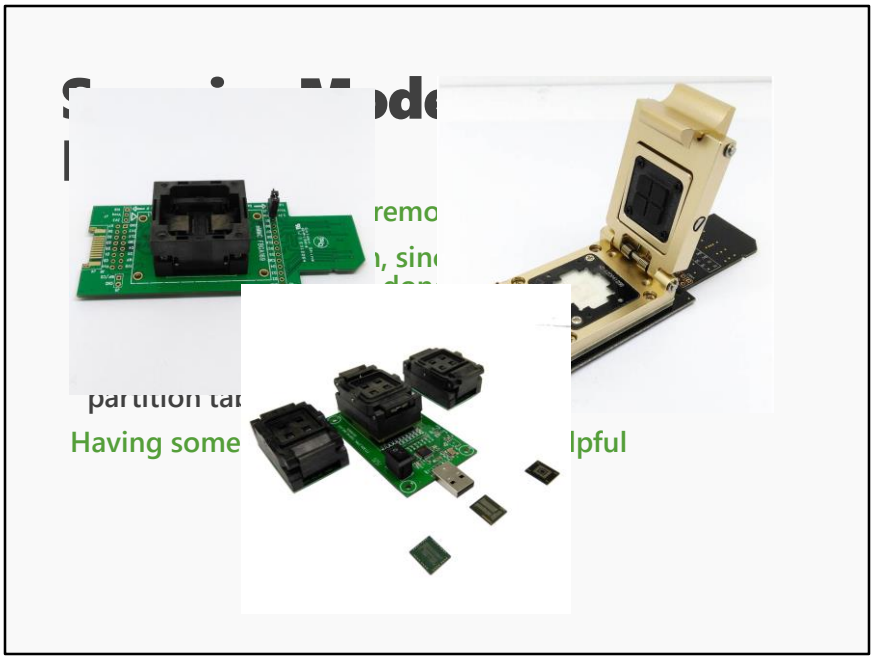
Surprise Mode: Direct Extraction

- Could be as simple as "remove SD card, image it"
- eMMC is harder though, since you need to get to the data lines, but it can be done!
- You will need to understand how the disk is laid out
  - Binwalk can help later, as can "standard" DOS partition tables.
- Having some in-vivo information is helpful

eMMC devices make things harder though. eMMC is really similar to SD in that they share a common ancestor, there's a few divergences that have been made to make the phone manufacturing industry happy.



More eMMC modules now are actually eMCP modules, where you have the eMMC module stacked in silicon alongside LPDDR.

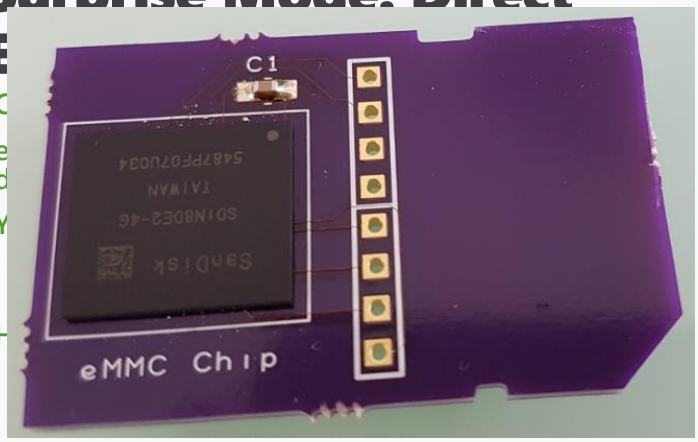


eMMC readers are readily available. They come in all shapes and sizes and even have USB versions. eMMC cards still talk the same protocol that MMC cards talk, and are thus compatible with SD cards!

Surprise Mode: Direct E

C
e
d
Y

H



http://blog.oshpark.com/2017/02/23/retro-cpc-dongle/

This also means you can be like this crazy bastard and make your own SD card adapters for eMMC devices.



eMMC devices are also more common in industrial applications, as their lack of a physical interconnect means they can be potted and conformal coated. They survive higher temperatures, more reads and writes than your typical consumer SD card, and are generally meant for more abuse.

These devices here are just two examples of Raspberry Pi clones – compatible ones, even – which are built around eMMC devices.

Surprise Mode: Direct Extraction

- Could be as simple as “remove SD card, image it”
- eMMC is harder though, since you need to get to the data lines, but it can be done!
- You will need to understand how the disk is laid out
  - Binwalk can help later, as can “standard” DOS partition tables.
- Having some in-vivo information is helpful
- All Else Fails: Solder to the rescue
  - Might need to desolder some storage, or otherwise physically attack the hardware
  - MTD devices are weird. Prepare to get your hands dirty
  - Interested in more? HHV and friends are the place to start looking.
  - JTAG, etc. might be the hard way out.

In the end, you may have to go out of your way and start sniffing around the device to try and find what’s going on. Your device might need a whole bunch of fun.

If you think this could be you, It’s time to go to the Hardware Hacking Village and learn you a Thing. Hardware is a whole different special chunk of fun that we won’t get into because that rabbit hole is deeper than some young men at a street fair in late September.

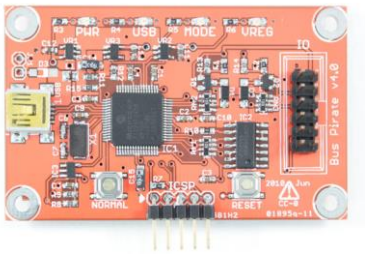


Logic Analyzers

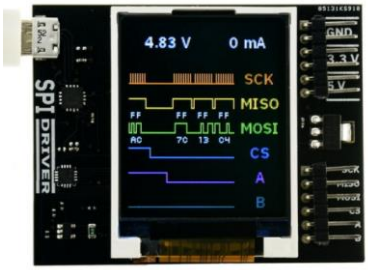
Salae makes a good one

- Cheap
- Basic
- Runs over USB

It might be time to get a logic analyzer. There's plenty of cheaper, USB based ones that are more than capable .



http://dangerousprototypes.com/docs/Bus_Pirate



https://www.crowdsupply.com/excamera/spidriver

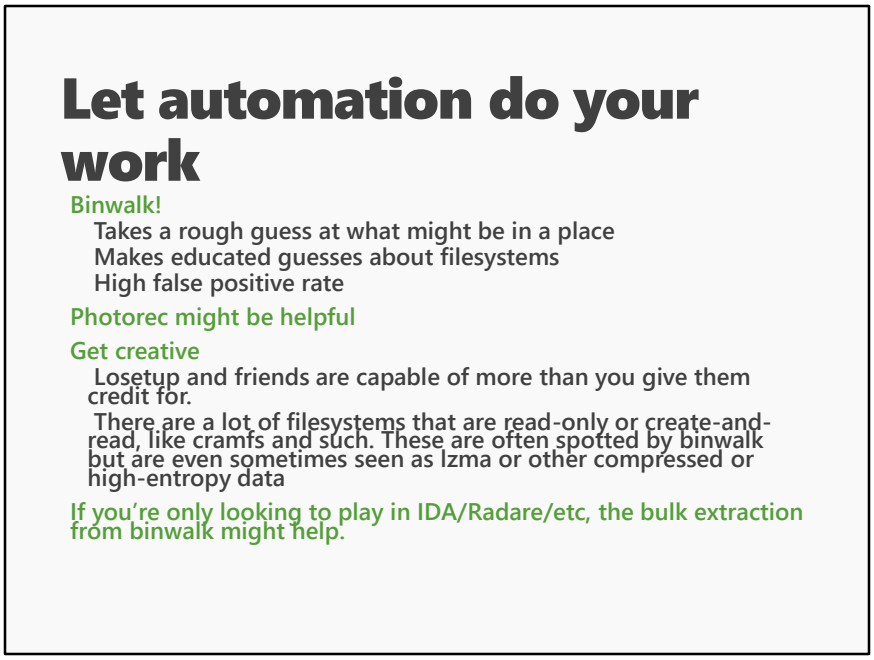
Hardware interfaces

Devices like the Bus Pirate and SPIDriver are convenient hardware bridges for the wide, wide world of hardware peripherals that let you bridge the gap. Sure, laptops have SPI and I2C but they're not easily accessible as a platform. These sorts of devices are amazing ways for you to mumble with hardware.

Now that we have that, what do?

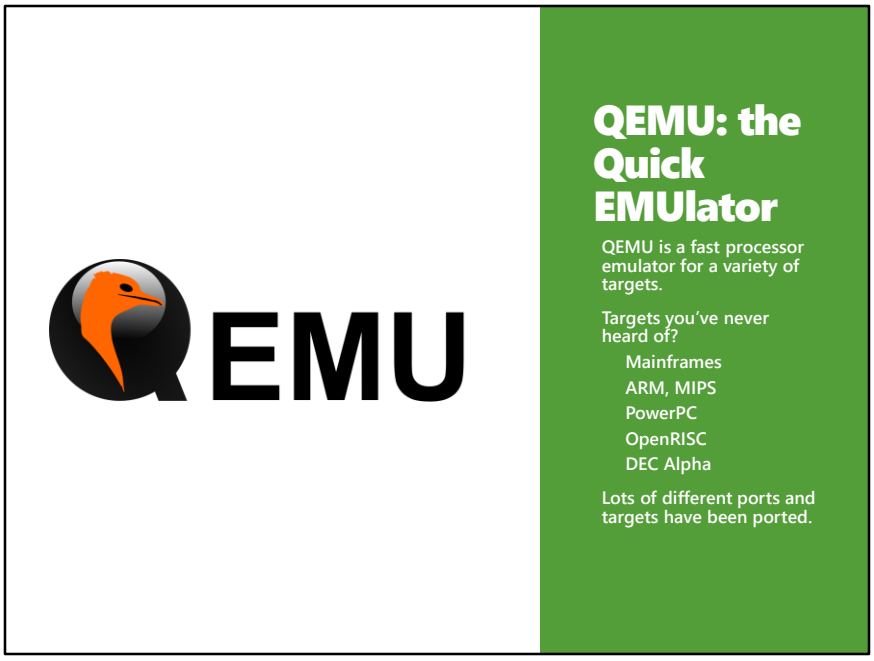
- Try mounting it/extracting it/etc. `file` might give you a good idea of what it thinks it might be, as will `strings` and the like.
- eMMCs sometimes have real partition tables
- SD cards often do
- Look at the reconnaissance you did
  - Boot logs: lots of good information about partitions
  - Fstab, /proc/mounts

So now we have the root filesystem, or a dump. It's time to start digging around.

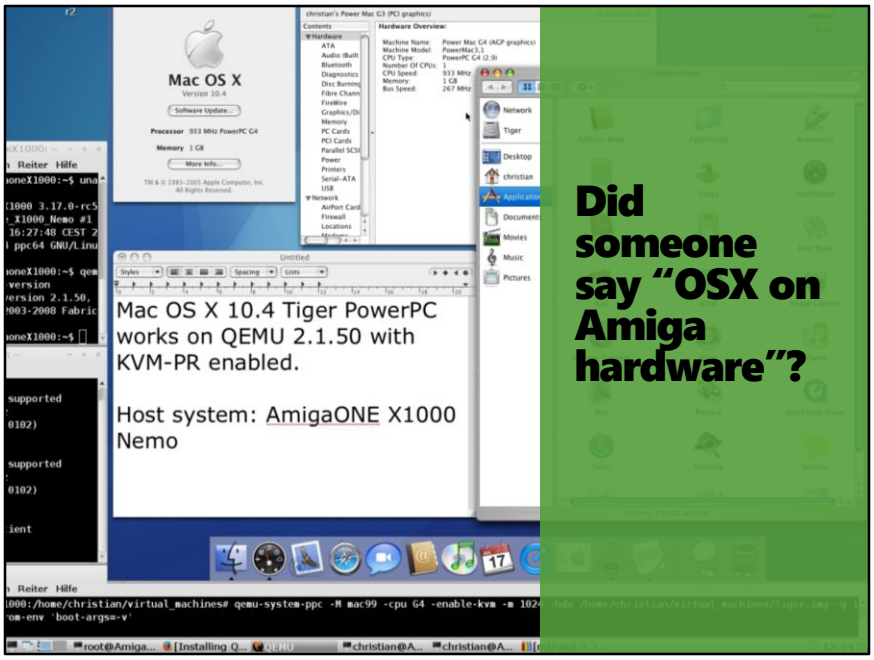


The biggest thing that you should get in the habit of is finding automation tools that work for your use case. Binwalk is the classic tool for this. It makes rough, educated guesses and spits out more guesses. It has a fairly high false positive rate, but a fairly low false negative rate, which makes it fairly useful in finding what could be hiding under there, so long as you can dig through the dirt.

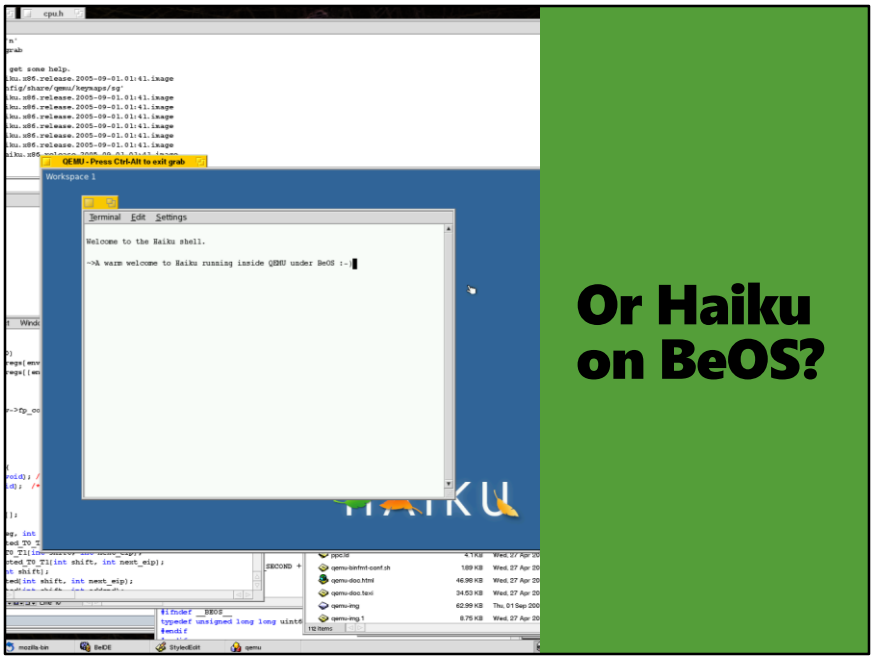
Tools like Photorec, losetup, etc. are sometimes useful in their own right: These sorts of tools have a little bit of smarts baked in that can be really helpful. There are compressed filsystems that binwalk will ignore that other tools will totally be happy to consume. Your worst case is that a tool makes your working directory a bit of a mess.



However, this brings us to one amazing chunk of automation: QEMU. QEMU is a fast emulator for a variety of different platforms, letting you do stupid shit like

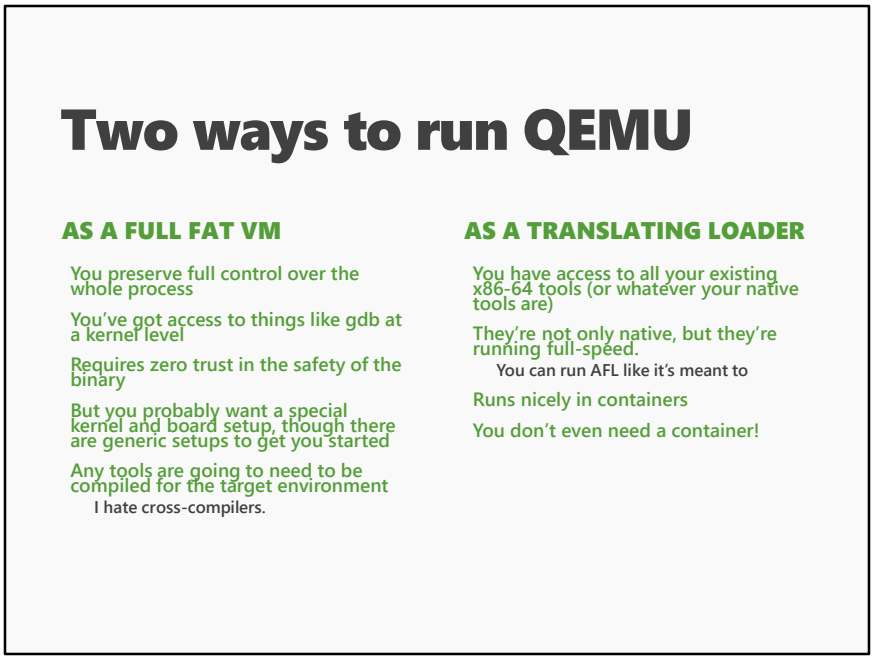


Run Mac OSX 10.4 on an Amiga



Or run Haiku on BeOS.

Or, alternately, more practical things.



QEMU has two general modes: A full-fat VM, like we just saw, or as a translating loader in Linux. Your device turned out to be X86 but you're a paranoid person and don't feel like letting it touch the real hardware directly? Full fat VM. Hell, even PowerPC can get in on the game. As a loader, it makes those weird executables from that hot new device look like any other executable.

Full-Fat VM: 9 tracks of DOS

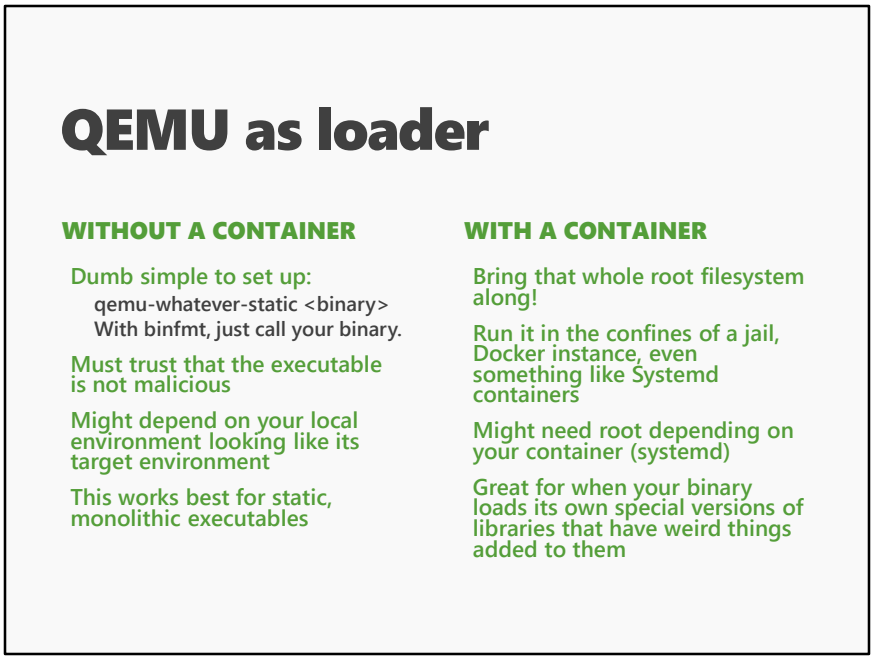


Here we see QEMU running as a full VM, running MSDOS 6.22 and interfacing with real hardware, a gigantic Overland Data 9 track tape drive. I used this to dump Data General AOS boot and install tapes.

Binfmt: Linux' way of loading executables

- Long ago, Linux added loadable loaders
- Originally for the purposes of running JARs from the command line like God and Sun Microsystems intended.
- Turns out this is a great place to put emulators.
- Debian ships with support for this in its *binfmt-misc* package.
- QEMU can be shipped as a "static userspace" environment (think WINE)
- Uses "magic numbers" – signatures from a database – to determine which ones are supposed to load what.
- Fairly simple to add new kinds of binaries. You could actually execute JPEGs if you really wanted to?

So let's talk about Linux's loader. Ever wondered why you can call Mono and WINE executables directly from the command line? Simple, it's the loader framework. Originally so that you could call JARs directly, it's now fairly trivial to add new kinds of "executables" to Linux.

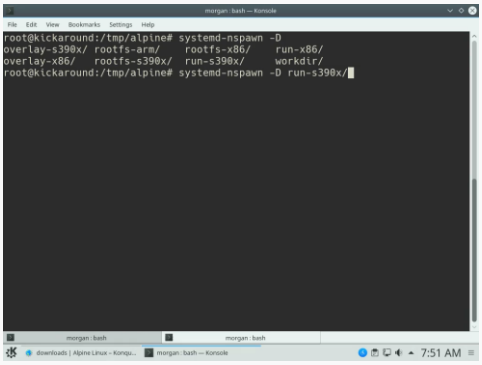


As a loader, it comes down to two options: With or without containers. Without a container, you're going in raw with no protection. If that executable decides to do something terrible to you, it's totally going to do it. On the other hand, containers are a totally reasonable option. Using containers means that you have some amount of wall around a possibly malicious application, or something that needs weird library setups. As long as Linux can call the loader, it's all fine.



To give you context on what you're going to see next, this is the hardware most software expects. Not a laptop running a VM running QEMU.

QEMU user Demo



Here we see QEMU running S390x executables from Alpine Linux, doing a full boot, then just running a shell from the ArmArch64

AFL setup

Oh boy. Let's talk about AFL.

AFL needs to compiled with QEMU support

Magic sauce: CPU_TARGET=whatever ./build_qemu_support.sh

AFL needs to bring along the host's libraries

Easiest bound with systemd-nspawn

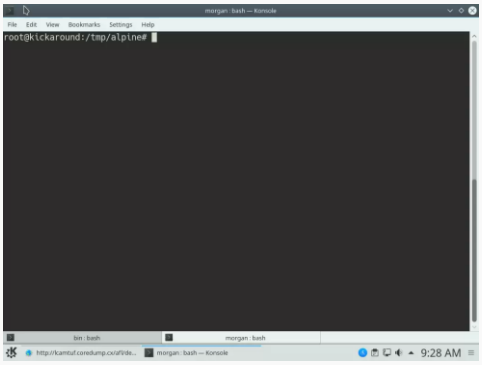
Don't do this in a VM

It hurts

In my outline, I promised a demo of AFL.

AFL needs a bunch of setup. I had to compile AFL under Debian, pull over a bunch of libraries from the local side, then pass in some environment arguments. Why am I going to these great lengths? I'm running it into the container that all the s390x executable lives in, which is under Alpine's version of musl libc.

AFL Demo


morgan : bash — Konsole
File Edit View Bookmarks Settings Help
root@ktckaround:/tmp/alpine#
bin : bash
morgan : bash
http://kamtuf.coredump.cx/afl/de...
morgan : bash — Konsole
9:28 AM

Wrapping up

What did we learn today?

- Hardware vendors are lazy
- Attacking hardware means getting creative
- QEMU is pretty neato
- AFL runs really slow when you're emulating an X86 emulating an IBM mainframe.

Going forward:

- I hope I've given you some idea of the landscape of tools
- Always remember rule 0

More Resources

Non-Linux targets:

RECON 2010 with Igor Skochinsky: Reverse Engineering for PC Reversers: http://hexblog.com/files/recon%202010%20Skochinsky.pdf

JTAG Explained: https://blog.senr.io/blog/jtag-explained

https://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-dehaas/bh-eu-04-dehaas.pdf

https://beckus.github.io/qemu_stm32/ (among others)

Linux targets:

eLinux.org – *Fucking Gigantic* wiki about embedded Linux.

linux-mips.org – Linux on MIPS

Thank you

Keep on hacking.