Understanding, Building and Researching Minimal (and not so minimal) Linux Systems

Ron Munitz
Managing Director - PSCG Holdings LTD
Founder & CEO - The PSCG

ron@thepscg.com

Open Source Summit / Embedded Linux Conference
San Diego, 2019

@ronubo
@thepscg
About Ron Munitz

• Managing Director of PSCG Holdings Ltd.
  – House of excellence for several embedded, IoT and Cyber-Security ventures.
  – House of The PSCG Premium Consulting and Training group

• Founder, CEO and Training Lead of The PSCG
  – The Premium Embedded Security, Linux and Android consulting firm
  – Authorized Training Partners of The Linux Foundation in APAC, ARM,
  – One stop shop for all of your offensive and defensive security concerns.

• Founder and (former) CTO of Nubo Software
  – The first Remote Android Workspace

• Adjunct Professor in several Academic Institutes
  – Next semester or two in Singapore (?)
About The PSCG

• The Premium Embedded Security, Linux and Android consulting firm
• One stop shop for all of your offensive and defensive security concerns.
• One stop shop for all turnkey solutions of Board Support Packages, Application Development and advanced debugging, optimizing and reverse engineering tasks.
• Premium Embedded, Security, and Data Science Training
• For more info: https://www.thepscg.com/
• Contact information:
  – Training: training@thepscg.com
  – Cyber Security academies: academy@thepscg.com
  – Consulting, and all other inquiries: info@thepscg.com
About The PSCG

Some of our customers:

- Samsung
- Qualcomm
- Cisco
- Intel
- Amazon
- Mellanox Technologies
- Bosch
- ARM
- Linux Foundation
- U.S. Air Force
- Ministry of Home Affairs
- MAFAT
- Japan's Ministry of Defense
- Oracle
- Arris
- Capital One
- Coca-Cola
- HP
- Hike Messenger
- Avast
- Marvell
- Cellebrite
- IAI
- Rafael
- SolarEdge
- Square
- IBS Software
- Amdocs
- Argus Cyber Security
- CyberBit
- CyberX
About PSCG Holdings LTD

● We are a super-group of low profile, highly skilled embedded security, virtualization and Operating System internals experts, accompanied by some of the brightest Data Science and Machine Learning experts around.

● Consulting and training is done via The PSCG and sponsors our ventures in diverse fields such as:
  ○ Mobile Security (platforms, apps and monetization)
  ○ Maritime and Aerospace Security
  ○ Multi display automotive deployments
  ○ More: less about security, more about making a positive impact
About PSCG Holdings LTD

- **Premium Training**
  - Short term courses and lectures (several days)
  - Long term government grade academies (several months)
- **Premium Consulting, Outsourcing and technical Due Diligence**
- **Penetration Testing, Security Hardening and Security Research**

Supports and Funds:

Startup Ideation and Development (PSCG Holdings umbrella)
About Our Gov/Mil Grade Academies

The PSCG provides the following, long term fully fledged training academies:

- Android/Linux/IoT Cyber-security Research
- Industrial Control System (ICS/SCADA/IloT) Cyber-security Research
- MacOS/iOS Cyber-security Research Research
- Windows Cyber-security Research Research
- Data Science and Machine Learning
Ready?

Getting Ready
Getting ready - required packages

- Setting up on your own machine:
  /* instructions are for 18.04.02 LTS which you can download from here */
  /* You can use a distro of your choice, and adjust the instructions accordingly */
  Install the following packages:
  
  $ sudo apt-get install make gcc flex bison \
  libssl-dev \
  libelf-dev \
  vim git tree \
  qemu-system-x86 \
  gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential \
  chrpath socat cpio libncurses5-dev \
  python python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping \
  python3-git python3-jinja2 libegl1-mesa libssl1.2-dev xterm \
  linux-tools-$(uname -r) bpfcc-tools

  For the ARM research courses also install the following packages:
  
  $ sudo apt-get install rpcbind \
  qemu-user binfmt-support qemu-user-binfmt \
  gcc-arm-linux-gnueabi gdb-multiarch \
  gcc-aarch64-linux-gnu \

  sudo usermod -aG kvm $USER
[Getting ready - provided VM]

- You can alternatively use the following Kernel development setup VM
  - Optional. requires VMware workstation player/fusion. It's better to work on your own environment if you can - nothing we do here is "not safe"
  - This is a Google Drive download - if you are in China/can't access → send me an email and I will help
  - When requesting access - you will be asked a couple of questions, so basically when you ask for access just tell about yourself, assure that you are attending the workshop, and I will respond with some instructions.
Getting Ready - "source code"

- Demo code we present in this tutorial:
  https://github.com/ronubo/OSS-Aug-2019
  - We talk about everything, but there are some trivial scripts that will help you refresh your memory as you go
  - We also show things that are not there (or in the slides) - we only have "that much" time - and as long as everyone is following class and terminal - we don't really need any of these slides.

- Binaries for this tutorial:
  - If you just want to run with us but not build with us
  - Just a kernel and ramdisk. No modules/other userspace executables, and not yocto project
Getting ready before class (will be deleted in published slides!)

Assuming you attend the class and want to have the right tarballs downloaded and make sure your environment works - please run the following (except for the [ ] line) before class. It will download the same busybox and kernel versions the instructor uses, and will also build them for you (so it may take time)

```bash
./fetch_kernel_tarball.sh
./build_kernel.sh
[./run_kvm.sh # but remove "-initrd ..."]
./fetch_busybox_and_create_initial_ramdisk.sh
./run_kvm.sh
```
Set...

How we're going to roll things
Methodology

- Our objective is to get you (at least those who setup their environments prior to the workshop) doing and running with the instructor.

- But time is short, and we need to cover at least some important theory to understand what and why we're doing what we're doing.

- So we will be more practical than rigorous:
  - For more detailed discussion → Contact the speaker after class.
Go!

Let's get to business
Part I - Intro

Getting the source
Getting Started
Simplified Kernel Boot Procedure
Part I - Intro

● Getting the source and getting started
  ○ Versions and variants
  ○ Kernel development setup - local
  ○ Source tree organization

● Boot process
  ○ Ramdisk, rootfs
  ○ The init process
Getting the source from kernel.org

The following screenshot (taken June 4th, 2019) shows nicely the concepts of a stable kernel going "EOL"
Getting the source from kernel.org

The following screenshot (taken at the day of the talk, August 22nd, 2019) shows the progress since the previous screenshot (2.5 months time span)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td><a href="https://www.kernel.org/pub/">https://www.kernel.org/pub/</a></td>
</tr>
<tr>
<td>Git</td>
<td><a href="https://git.kernel.org/">https://git.kernel.org/</a></td>
</tr>
<tr>
<td>RSYNC</td>
<td>rsync://rsync.kernel.org/pub/</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Protocol</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainline</td>
<td>5.3-rc5</td>
<td>2019-08-18 [tarball] [patch] [inc. patch] [view diff] [browse]</td>
</tr>
<tr>
<td>stable</td>
<td>5.2.9</td>
<td>2019-08-16 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>stable</td>
<td>5.1.21 [EOL]</td>
<td>2019-07-28 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>longterm</td>
<td>4.19.67</td>
<td>2019-08-16 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>longterm</td>
<td>4.14.139</td>
<td>2019-08-16 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>longterm</td>
<td>4.9.189</td>
<td>2019-08-11 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>longterm</td>
<td>4.4.189</td>
<td>2019-08-11 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>longterm</td>
<td>3.16.73</td>
<td>2019-08-20 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse] [changelog]</td>
</tr>
<tr>
<td>linux-next</td>
<td>next-20190822</td>
<td>2019-08-22 [tarball] [pgp] [patch] [inc. patch] [view diff] [browse]</td>
</tr>
</tbody>
</table>

Latest Stable Kernel: 5.2.9
git: cloning a kernel tree

Obtaining a tree:

- Stable tree:
  
  $ git clone \\

- Shallow clone from linus's tree
  
  git clone --branch v5.3-rc3 --depth 1 \\
  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
git: Figuring out available tags

- On an existing tree (without having to `pull` or `fetch`):
  
  ```
  $ git ls-remote --tags
  
  $ git ls-remote --tags \
  ```

- Outside of a git tree:

**Exercise:** Can you tell what is the latest stable version of Kernel 4.x?
git: Catching up with the stable tree

- On an existing tree (pull or fetch):

  $ git pull

  # Done 16 June 2019, a day 5.1.10 was out, and the tree was synced to 5.1.9

remote: Compressing objects: 100% (906/906), done.
remote: Total 1973 (delta 1410), reused 1627 (delta 1064)
Receiving objects: 100% (1973/1973), 2.42 MiB | 463.00 KiB/s, done.
Resolving deltas: 100% (1410/1410), done.
  2bf3258..a74d0e9  linux-4.14.y -> origin/linux-4.14.y
  768292d..7aa823a  linux-4.19.y -> origin/linux-4.19.y
  2df1614..7e1bdd6  linux-5.1.y -> origin/linux-5.1.y
* [new tag]         v4.19.51   -> v4.19.51
* [new tag]         v5.1.10    -> v5.1.10
You are not currently on a branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

  git pull <remote> <branch>
Kernel Versions

When a Kernel is declared stable, a Merge Window begins, where features from Linux Next would be selected into the next prospective mainline kernel version.

Then, a series of Release Candidates (RC) starts, in its end, a new stable kernel would be declared.

Booting Linux

- In PC’s as well as in embedded architecture, sooner or later a **boot loader** will be loaded into memory from some persistent storage such as flash (mostly in embedded systems) or hard drive.
- In the Embedded world, it is common to use **Das U-Boot**
- On the PC and Server world, **GRUB 2** is the de-facto standard for Linux booting
  - But there are other bootloaders such as **systemd-boot** *(rip gummiboot)*, ancient **LILO** and more
- The objective of both: Load a **Linux Kernel** image into memory and boot it
Booting Linux on X86

- Sooner or later (in a chain of bootloaders) a **bootloader** will be loaded from a boot device. We will concentrate on **GRUB** as it is the most common bootloader:
  - The boot device would usually be an installed hard drive or a USB device
  - In legacy BIOS it is loaded from a small loader that is written on the disk MBR sector
  - In UEFI it is loaded from the ESP partition and is essentially an EFI program.
- GRUB will then look at boot configurations, and will try to locate a **kernel** and **initial ramdisk** [and device tree]
- Then, it will load the kernel to the memory, and pass it its **command line**

**Note:** the only thing that matters is the kernel. You will see configurations with device tree blobs, rootfs and more. initramfs is common in the PC world

**Note:** It is also possible to use GRUB to boot from network!

**Note:** It is also possible to boot Linux directly from UEFI (see `CONFIG_EFI_STUB`)
Booting Linux on ARM

- On power-up, a boot configuration is selected on the main CPU by a set of *bootstrap resistors* and *eFuses*
- The boot CPU will have an exception, for which it will execute *bootrom/initial bootloader* code
  - from some non-volatile memory (e.g. on SOC ROM),
  - via some well defined method (e.g. via SPI)
- The code will then obtain an additional *bootloader* which will setup DRAM, configure peripherals, and start another set of bootloaders, some executable image, or an Operating System
  - e.g.: U-Boot $\rightarrow$ Linux Kernel
ARM/U-boot kernel boot example

- Press any key to stop the boot process at U-Boot prompt
- `U-boot> help`  # see commands
- `U-boot> setenv bootargs` “console=ttymxc0,115200 thePSCG=rocks
- `U-boot> loadb 0x10800000` # load kernel binary over serial line not sure we need it. This is the default address for loadb anyway
- `U-boot> loadb 0x10100000` # load filesystem
- `U-boot> loadb 0x10200000` # load device tree
- `U-boot> bootm 0x10800000 0x102… 0x120…` # Define vendor addresses
- Now wait until the login and play around with your Linux system
- Explain persistency (`uboot.env…`)
Linux Kernel Initialization

- Once the Linux Kernel has loaded, it will:
  - [In most cases unpack itself (@see piggy et. al)]
  - Do specific architecture dependent initializations, including moving modes (e.g. \textit{real} \rightarrow \textit{protected})
  - And then jump to an architecture independent \texttt{start_kernel()} function at \texttt{init/main.c}
    - Which will eventually jump back and forth between arch dependent/independent code
  - Spawn the \texttt{kernel_init()} thread
    - Which will look for an \textit{init} executable depending on what was given in the command line - and literally \texttt{execve()} it in a less privileged mode - that is the entry to userspace.
  - Spawn \texttt{kthread}…
Linux Userspace Initialization

- Once the Linux Kernel has loaded, it will
  - ...
  - Spawn the `kernel_init()` thread
    - Which will look for an `init` executable depending on what was given in the command line - and literally `execve()` it in a less privileged mode - that is the entry to userspace.
    - If it can’t find one, or it exits - the system will `panic()`.
  - ...
- From this point and on, “Linux” diverges.
- Each implementation can do whatever it wants although there are common `init` frameworks
  - The (unofficial) standard for all modern distros is `systemd`, which is the system startup and services management component
  - Other frameworks include `upstart` and `SysV init`
  - You can implement your own, or just start a `busybox` executable.
  - Another notable example is Android’s on init (@see `system/core/init`)
Rootfs Motivation (simplified)

- When the kernel boots, it needs to be aware of the concept of a filesystem - namely some place to represent, and access files, such as the init file, which as we will see later, is what "defines" the user space behavior.

- A file system effectively defines a series of well known "operations", many of them are nicely exported to userspace to transparently provide access via system calls such as open, read, write, lseek, ioctl etc.

- This is nicely presented in the kernel via an abstract concept of a Virtual File System (VFS) which effectively has handles (function pointers) to concrete file systems whether they are on a real device, on memory etc.

- So a legit, and trivial file system would be one that is completely represented in memory and present a virtual concept of files
Without getting into the technicalities of different memory file systems (e.g. tmpfs, ramfs, etc), the rootfs is an instance of an in-memory file system that is always present, and is mounted at /.

Most systems will just mount another filesystem over the initial rootfs and change root (so the previous one is "gone.")

- There are significant subtleties here:
  - chroot (changes directory and runs init - escapable by exit)
  - pivot_root (saves the previous root directory)
  - switch_root (ditches everything and merciless changes directory and runs init)

Nuances: chroot vs. mount namespaces (e.g. "containers")

- Experience shows that here at least 10 different people would want to ask "Docker questions". Not for this talk - time is pressing
- Ask me later and I'd be happy to discuss and explain security boundaries/implementations/etc

The initial filesystem to be provided to the kernel will usually be denoted either ramdisk or initramfs (and there are differences between the two)
initramfs

- The only purpose of an initramfs is to mount the root filesystem (or be the root filesystem)
- **initramfs** is a complete set of directories that you would find on a normal root filesystem.
  - It is usually bundled into a *cpio* archive
  - The *cpio* archive is usually **compressed**
- At boot time, the bootloader loads the kernel and the initramfs image into memory and starts the kernel.
- The kernel checks for the presence of the initramfs and, if found, mounts it as / and runs `/init`.
- The *init* program is typically a shell script (the kernel knows to interpret them as executables!)
  - To understand why - see *binfmt*
- **Note that the boot process may take longer if an initramfs is used.**

Note: initramfs can be baked into the kernel image itself (quite common in embedded)
For most distributions, **kernel modules** are the biggest reason to have an initramfs.

- In a general distribution, there are many unknowns such as file system types and disk layouts.
- For some embedded systems, the different storage layouts are known and a custom kernel is normally built - so an initramfs may not be needed.
- In a modern distro (such as Ubuntu) you would see that the **initramfs** is packed a bit differently (use `unmkninitramfs` to unpack) and contains modules, some executables (e.g. run-init) and architecture microcode files.

Otherwise, there are five primary reasons to have an **initramfs**:

- loading the rootfs from a network
- loading it from an LVM logical volume
- having an encrypted rootfs where a password is required
- or for the convenience of specifying the rootfs as a LABEL or UUID. Anything else usually means that the kernel was not configured properly.
- Using the Linux boot process to chroot/switch_root to another instance (e.g. Android-X86)
init and users

- Upon initialization, the init process runs as a privileged user (root) in a non-privileged mode (user mode).
- An operating system can operate without users.
- It could also operate perfectly well (and even slightly more efficiently!) with one mode (e.g. have each task access another task’s memory space).
- Actually the world could work without doors and locks.
  - Unfortunately, there are both bugs and security vulnerabilities in human nature...
  - So security mechanisms are needed.
Q&A: Init and Users

- Q: What makes a privilege?
  - A: Access Control

- Q: Who governs access?
  - A: Access is governed in multiple levels:
    - Application level
    - Trusted Daemons
    - Kernel Level
    - Trusted Entities that monitor it
    - Trusted External Hardware
      - TEE, SE, HSM’s, Accelerators, External Appliances
[Ubuntu boot process main steps]

● BIOS/EFI (let's assume EFI)
● GRUB (e.g. `grubx64.efi`)
● Kernel and initramfs are loaded to memory
● Kernel is boot
● Init process (PID 1) is started
  ○ Running `init` from init ramfs
    ■ This is a shell script → for which the interpreter is `busybox`
  ○ Long story short: mounts a hands full of stuff and then calls `run-init`

/* if time permits, show the files */
$ tree -L 2

.  ├── early
│     └── kernel
└── main
     ├── bin
     ├── conf
     ├── etc
     ├── init
     ├── lib
     ├── lib64
     ├── run
     ├── sbin
     ├── scripts
     ├── usr
     └── var

Interesting files to note:

- main/init
- main/bin/run-init
  @see https://git.kernel.org/pub/scm/libs/klibc/klibc.git/tree/usr/kinit/
- main/bin/busybox
Part II - QEMU

Building mini_linux (busybox+vanilla kernel)
Building kernel modules
Building userspace
Debugging and other tips and tricks
Part II - QEMU

- Setting up a minimal working Linux distro
  - QEMU/KVM based
  - With virtually zero effort

- Kernel debuggers
  - kdb
  - Hardware debuggers (in qemu)
  - Cross gdb/kgdb
Source/Scripts

./fetch_kernel_tarball.sh
./build_kernel.sh
[./run_kvm.sh # but remove "-initrd ..."]
./fetch_busybox_and_create_initial_ramdisk.sh
./run_kvm.sh

Explanation: this is for the lazy. always do things manually and fail to understand what is what.
Building a Kernel

- Change directory to your kernel source
  $ cd ${KERNEL_SRC}

- Create a .config file
  $ make O=${KERNEL_OUT} defconfig

- Add to .config the relevant kvm configs
  $ make O=${KERNEL_OUT} kvmconfig  # this also runs scripts/kconfig/merge_config.sh !

- Build the kernel
  $ make -j8 O=${KERNEL_OUT}
Running the kernel

- After building, you will have at `${KERNEL_OUT}`:
  - `vmlinux` - kernel output with debug information
  - `arch/${ARCH}/boot` - stripped and compressed kernel

- What happens if you run either of them?
Building a minimal ramdisk

- **Get busybox:**
  
  ```
  $ wget https://busybox.net/downloads/busybox-1.30.1.tar.bz2
  $ tar xf busybox-1.30.1.tar.bz2
  ```

- **Configure and build busybox**
  
  ```
  $ cd busybox-1.30.1/
  $ make defconfig
  $ sed -i 's:# CONFIG_STATIC is not set:CONFIG_STATIC=y:' .config
  $ make -j16 CONFIG_PREFIX=../wip_ramdisk install
  $ cd ..
  ```

- **Create an initial directory structure**
  
  ```
  $ mkdir wip_ramdisk/{lib,proc,sys,dev,mnt,xbin}
  ```

- **Populate init script**
  
  ```
  #mounts, prints, and...
  exec /bin/sh
  ```
Running an initrd

Now that we have an *initrd* you can try to experiment with several things e.g.:

- What happens if you don't have an *init* script/executable?
- What happens if you have a *busybox* based init, that does not have *execution* permission?
  ■ Who enforces that?

Do try that at home!
Installing in-tree modules

- Install the built kernel modules (*.ko):

  
  $ make  O=${KERNEL_OUT} INSTALL_MOD_PATH=${RAMDISK_WIP} modules_install

make[1]: Entering directory '/home/ron/OSS-Aug-2019/linux-5.3-rc3-qemu-x86_64-out'
  INSTALL crypto/crypto_engine.ko
  INSTALL drivers/crypto/virtio/virtio_crypto.ko
  INSTALL drivers/thermal/intel/x86_pkg_temp_thermal.ko
  INSTALL fs/efivarfs/efivarfs.ko
  INSTALL net/ipv4/netfilter/iptable_nat.ko
  INSTALL net/ipv4/netfilter/nf_log_arp.ko
  INSTALL net/ipv4/netfilter/nf_log_ipv4.ko
  INSTALL net/ipv6/netfilter/nf_log_ipv6.ko
  INSTALL net/netfilter/nf_log_common.ko
  INSTALL net/netfilter/xt_LOG.ko
  INSTALL net/netfilter/xt_MASQUERADE.ko
  INSTALL net/netfilter/xt_addrtype.ko
  INSTALL net/netfilter/xt_mark.ko
  INSTALL net/netfilter/xt_nat.ko
  DEPMOD  5.3.0-rc3
make[1]: Leaving directory '/home/ron/OSS-Aug-2019/linux-5.3-rc3-qemu-x86_64-out'
What happens after `modules_install`

```bash
$ tree ${RAMDISK_WIP}/lib
/home/ron/OSS-Aug-2019/wip_ramdisk/lib
  └── modules
      └── 5.3.0-rc3
          └── build -> /home/ron/OSS-Aug-2019/linux-5.3-rc3-qemu-x86_64-out
          └── kernel
              └── crypto
                  └── crypto_engine.ko
              └── drivers
                  └── crypto
                      └── virtio
                          └── virtio_crypto.ko
                  └── thermal
                      └── intel
                          └── x86_pkg_temp_thermal.ko
              └── fs
                  └── efivarfs
                      └── efivarfs.ko

...
What happens after `modules_install`

```
$ tree ${RAMDISK_WIP}/lib
/home/ron/OSS-Aug-2019/wip_ramdisk/lib
  └── modules
      └── 5.3.0-rc3
          └── build -> /home/ron/OSS-Aug-2019/linux-5.3-rc3-qemu-x86_64-out
              └── kernel
                  └── net
                      └── ipv4
                          └── netfilter
                              └── iptable_nat.ko
                              └── nf_log_arp.ko
                              └── nf_log_ipv4.ko
                      └── ipv6
                          └── netfilter
                              └── nf_log_ipv6.ko
                          └── netfilter
                              └── nf_log_common.ko
                              └── xt_addrtype.ko
                              └── xt_LOG.ko
                              └── xt_mark.ko
                              └── xt_MASQUERADE.ko
                              └── xt_nat.ko
```

Continued in next slide
What happens after `modules_install`

```bash
$ tree ${RAMDISK_WIP}/lib
/home/ron/OSS-Aug-2019/wip_ramdisk/lib
    └── modules
        └── 5.3.0-rc3
            └── build -> /home/ron/OSS-Aug-2019/linux-5.3-rc3-qemu-x86_64-out
                └── modules.alias
                └── modules.alias.bin
            └── modules.builtin
                └── modules.builtin.bin
            └── modules.builtin.modinfo
            └── modules.dep
            └── modules.dep.bin
            └── modules.devname
            └── modules.order
            └── modules.softdep
            └── modules.symbols
                └── modules.symbols.bin
            └── source -> /home/ron/OSS-Aug-2019/linux-5.3-rc3
```

We can see several interesting links (link to source, and link to build folder) and files (module dependency information)
What happens after `modules_install`

Now you can:

- Rebuild your ramdisk
- Re-run your kernel+ramdisk
- Insert one of the built `.ko`'s
  - Either using `insmod` with the full path
  - Or by using `modprobe` with the basename of the `.ko`
    - Remember those module dependency files?

So now we have in-tree loadable kernel modules covered.

What happens if we want to build Out-Of-Tree modules?
Building out of tree modules

- Build the kernel modules
  - Providing the module Makefile to specify
    ```shell
    obj-m += yourmodule.ko
    ```
  - And then using the kernel build system
    ```shell
    $ make -C ${KERNEL_OUT} M=$(pwd) modules
    ```

- Install the modules to the ramdisk (/rootfs)
  ```shell
  $ make -C ${KERNEL_OUT} INSTALL_MOD_PATH=${RAMDISK_WIP} modules_install
  ```
Adding some debug tweaks

- By one of the menu configuration tools
- By adding a *config fragment*
- We will use it from the provided repository
- Illustration of:
  - `scripts/kconfig/merge_config.sh`
    
    ```
    $ ${KERNEL_SRC}/scripts/kconfig/merge_config.sh -m
    kernel-configs/x86_64_defconfig_and_kvmconfig.config \ 
    kernel-debug-fragment.config
    
    $ mv .config kernel-configs/x86_64_kvm_debugtweaks.config
    ```

  - `scripts/diffconfig`
    
    ```
    ${KERNEL_SRC}/scripts/diffconfig ${KERNEL_OUT}/.config \ 
    kernel-configs/x86_64_kvm_debugtweaks.config
    ```
Building with the new changes

Several options:

- **Override `.config` in the build directory**
  
  ```
  $ mv ${KERNEL_OUT}/.config ${KERNEL_OUT}/.config.old
  $ mv kernel-configs/x86_64_kvm_debugtweaks.config ${KERNEL_OUT}/.config
  ```

  This only sets the configs - when done - build.
  
  ```
  $ make -C ${KERNEL_OUT} -j8
  ```

- **Set `KCONFIG_CONFIG`**
  
  But then you must manually approve new changes (or do another phase in which you first run the make command with the `olddefconfig` target) - and it may override your config files on the one hand, and will not overwrite the `.config` file on the other hand, so better not use it if you don't know what to expect!

  ```
  $ export KCONFIG_CONFIG=${LABS}/kernel-configs/x86_64_kvm_debugtweaks.config
  $ make -C ${KERNEL_OUT} -j8
  ```

  Note that `KCONFIG_CONFIG` requires a full path name - or a relative path to the kernel build directory

  - so if we gave it just `kernel-configs` it would look under the kernel build dir
  - And when it saw it was empty - would revert to using `.config`
Debugging and Tracing

- kdb demo
- kgdb stub demo
- [ftrace]
- [probes]

[] - As time permits. This time it probably won't permit
Part III - LKM's and basic forensics

- How to "inject new kernel code" by building and loading a loadable kernel module (LKM)
  - Hello World
  - [Introduction to device drivers]
  - [Addresses, symbols, and kptr_restrict]

- Basic forensics with virtual file systems
  - procfs
  - debugfs
  - sysfs and the Unified Device Model
Part IV - Security Mechanisms

- Hijacking system calls
  - Protections and subverting
- Kernel security starting points
  - certs
  - crypto
  - LSM's
  - ipfilter
  - BPF

Sorry, we have less than two hours - next time! :-(
So what do we have so far?

- Need to build kernel and features
- Need to build userspace tools
- Need to pack kernel and userspace tools
- Need to handle dependencies (e.g. package management?)
- Need to be easier to use and update
- Sounds like a distro (e.g. Debian, Redhat,...)

But what if we want full control? Or to support all kind of boards/deployments?

- **We need a build system to help us.**
  - ⇒ Yocto Project for the rescue
    - And so are other distros as well
Part III - Build Systems

Yocto Project
Building Linux Systems

- Building Linux Systems is a complex subject with many use cases
  - And countless corner cases
- However, the prerequisites, techniques (up to "how to specify what") and theory of operation are similar and are roughly divided into well known operations (see next slide) which aim to:
  - Prepare the **host environment**
  - **Specify what you want** to build and/or put on your host (tools) or target systems in a well known procedure (e.g. "recipes")
  - **Build software** for the target architecture and prepare the means to flash the **build artifacts**
Building Linux Systems

Common steps in building Linux Systems:

- Obtaining the source code of **cross-toolchains** and building them (or using prebuilt ones)
- Obtaining the source code for the **Linux Kernel** and potentially **bootloaders** for the specific target architecture and variants (including device trees if need be), **configuring** and **building** them
- Obtaining the source code [and pre-built tools] of **userspace packages** and building them
- **Packing** userspace tools into respective **root file systems**
- Specifying **Licenses** (it is compulsory for shipping!)
- **[packaging** (e.g. rpms, deb, etc.)]
- Creating the **final "images"** from the filesystems for flashing to the devices
- **Flashing** the systems (or providing means to flash them)

Building also includes configuring, which often uses well known tools such as **autoconf, automake, make, cmake, gradle** etc.
Once you have a built image, and flashed it, you would want to test it for sanity.

In most cases, the first thing you would want to do is to get a **console (serial access)**.
- Sometimes it would be with *kbd,kms* i.e., with a keyboard and a display.

It is up to the board designer, the bootloader configuration and the kernel configuration to enable the console on the target.

It is up to you to:
- Connect the cables to the right places in your host and the target, and verify the console parameters on the target (e.g. `console=/dev/ttyACM0 ...`) and on the host (e.g. `/dev/ttyUSB0`) are correct.
- Connect to the target via a terminal emulator (such as *minicom*, *miniterm.py*, *screen*, etc.).

Once you have a console - you are in the game!
- Of course, there are many, many challenges after that...
Embedded Development Overview

HOST
- Linux kernel (3.10.7.3)
- GNU toolchain
- GNU make (3.81)
- Python (2.7.3)-
- Shell (bash 4.2.25)
- Oracle JDK (1.6.0.34)
- Git (1.7.9.5)
- repo (1.12.2)
- Cross toolchains

TARGET
- BSP
- Kernel
- Drivers (?)
- Userspace (?)
- Shell (?)
- Graphics (?)
- ...

Android-X
- 86 over QEMU
- Windows over VirtualBox
- Linux over VmWare
- Android Emulator (X86)
- Android Emulator (ARM)
- LFS over UML
Embedded Build System - Overview

HOST

Operating system
Host toolchains
Cross toolchains
Source control

Build (I)
Images (Build Artifacts) for target (I)

Build (II)
Images (Build Artifacts) for target (II)

Linux-X8 6 over QEMU
Windows over VirtualBox
Linux over VmWare
Android Emulator (X86)
Linux Emulator (ARM)
LFS over UML
Embedded Build System - The Yocto Project way

HOST
- Linux kernel (3.10.7.3)
- GNU toolchain
- GNU make (3.81)
- Python (2.7.3-)
- Shell (bash 4.2.25)
- Oracle JDK (1.6.0.34)
- Git (1.7.9.5)
- repo (1.12.2)
- Cross toolchains

tmp/.../ kernel, rootfs,

bitbake (I)

bitbake (II)

tmp/.../ kernel, rootfs, dtb

Linux-X86 over QEMU
Windows over VirtualBox
Linux over VmWare
Android Emulator (X86)
Linux Emulator (ARM)
LFS over UML

Oracle JDK (1.6.0.34)
Git (1.7.9.5)
Yocto Project

- Yocto Project builds a distro for you
- It takes a series of rules (metadata) as textual files, and goes through the common phases of build systems such as:
  - Fetching
  - Configuring
  - Compiling
  - Packaging
  - Creating file systems and sysroots
  - [Sanity Checking/Testing]
  - Preparing images for deployment
Yocto Project Demo

- [Yocto Project ]
- [Creating your own tools]
- More info → YoctoProject.org and our courses.
Taking it from here
Next courses in San Diego and Singapore

I will be teaching the following public courses in San Diego:

- **ARM/Linux Kernel security research: Building, Debugging, Reversing, Forensics.** (August 19-21, 2019)
- **Android Internals for Cybersecurity Engineers and Platform Builders** (August 26-30, 2019)

Other relevant public courses in the next month in Singapore:

- Linux Reverse Engineering via Malware Analysis **(September 2-5)**
  - for x86 platforms
- ARM Exploitation **(September 17-20)**
- Yocto Project for Embedded Linux & IOT **(September 24-27)**

For more details and registration **ron@thepscg.com**
Backing slides - chroot demos
**Backing Slides - LXD rootfs demo**

- LXD rootfs extraction
- Creating an ext4 file system
- Providing it to qemu as a harddrive
- mountiing it
- changing root into it
- Chroot/switch_root/pivot_root discussion (see next slide)
  - challenges of pivot_root (cannot unmount initramfs)
  - Usage of the rest

- Nice demo/game: no idea what are the login credentials to the LXD rootfs for ubuntu 18.04.03. What do we do then?
  - chroot as a root
  - run passwd for the users we want to modify
  - profit!
Backing slides chroot and friends demo

- chroot and switch_root on initramfs
- Challenges with uid
  - i.e. if rootfs is not prepared properly and uses, e.g. user "ron" instead of root for chroot/switch_root/busybox etc.
- Complete switch_root scenario Booting systemd
For the sake of completion we present a quick demonstration of what makes 3 of the most popular (partially) open-source (OK, for the last one very partially) operating systems

Meaning that at least part of the functionalities of each of these operating systems can be worked with / ported to a similar workflow.

Let's discuss the current state of the art in emulation and debugging of such.
Backing slides - OS architecture
Linux Overview

Linux \(\Rightarrow\) (kernel)

- Executables/Daemons
- Display Managers, Window System, Graphical Apps,...
- Libraries
- syscall API
- Hardware
Android Overview

Linux Kernel - Not part of the Android tree per se.
- [https://www.kernel.org](https://www.kernel.org)
- [https://android.googlesource.com/kernel/](https://android.googlesource.com/kernel/)
- Vendor specific tree
[MacOS/iOS Overview]

- Executables/Daemons
- Apps
- Frameworks
- Libraries
- Low level access Libraries
  - Kernelcache /kernel
    - MACH
    - IOKIT
    - BSD
- Hardware